Journal on Satisfiability, Boolean Modeling and Computation 6 (2009) 165-201

# Lemmas on Demand for the Extensional Theory of Arrays

**Robert Brummayer** robert.brummayer@jku.at
**Armin Biere** armin.biere@jku.at
*Institute for Formal Models and Verification*
*Johannes Kepler University Linz*
*Austria*

## Abstract

The quantifier-free extensional theory of arrays $T_A$ plays an important role in hardware and software verification. In this article we present a novel decision procedure that refines formula abstractions with lemmas on demand. We consider the case where $T_A$ is combined with a decidable quantifier-free first-order theory $T_B$. Unlike traditional lazy SMT approaches, where lemmas are added on the boolean abstraction layer, our decision procedure adds lemmas in $T_B$. We discuss our decision procedure in detail. In particular, we prove soundness and completeness, and discuss complexity. We present our decision procedure in a generic context and provide implementation details and optimizations, in particular for bit-vectors. Finally, we report on experiments and discuss related work.

KEYWORDS: *SMT, arrays, bit-vectors, decision procedures*

*Submitted January 2009; revised April 2009; published May 2009*

## 1. Introduction

Arrays are important and widely used non-recursive data-structures, natively supported by nearly all programming languages. Software systems rely on arrays to read and store data in many different ways. In formal verification arrays are used to model main memory in software, and memory components, e.g. caches and FIFOs, in hardware systems. Typically, a flat memory model is appropriate and a memory is represented as a one-dimensional array of bit-vectors. The ability to compare arrays supports complex memory verification tasks like verifying that two algorithms leave memory in the same state. Reasoning about arrays is an important issue in formal verification and efficient decision procedures are essential.

Recently, Satisfiability Modulo Theories (SMT) solvers gained a lot of interest both in research and industry as their specific decision procedures, e.g. for the theory of arrays, turned out to be highly efficient. The SMT framework provides first-order theories to express verification conditions of interest. An SMT solver decides satisfiability of a formula expressed in a combination of first-order theories. Typically, specific decision procedures for these theories are combined by frameworks like [4, 45, 50]. Furthermore, theory combination does not have to be performed eagerly, but may also be delayed [15, 24]. If the formula is satisfiable, then most SMT solvers provide a model. In formal verification, a model typically represents a counter-example which may be directly used for debugging.

SMT approaches can be roughly divided into *eager* and *lazy*. In the lazy approach, the DPLL [28] procedure is interleaved with highly optimized decision procedures for specific

first-order theories. Decision procedures may be layered, i.e. fast (incomplete) subprocedures are called before more expensive ones [22].

Boolean formula abstractions are enumerated by the DPLL engine. Theory solvers compute whether the enumerations produce theory conflicts or not. Boolean lemmas are used to iteratively refine the abstraction. The DPLL procedure is responsible for boolean reasoning and is the heart of this approach. Even case splits of the theory solvers may be delegated to the DPLL engine [8]. For more details about lazy SMT see [46, 49].

In the eager approach, the theory constraints are eagerly compiled into the formula, i.e. an equisatisfiable boolean formula is built up front. For example, in the theory of uninterpreted functions, function symbols are eagerly replaced by fresh variables. Furthermore, congruence constraints are added to the formula. These constraints, which are also called Ackermann constraints [3], are used to ensure functional consistency. Finally, the formula is checked for satisfiability once. In contrast to the lazy approach, no refinement loop is needed. For more details about SMT and first-order theories see [13, 16, 40, 46, 49].

We present a novel decision procedure for the extensional theory of arrays. In our decision procedure an over-approximated formula is solved by an underlying decision procedure. If we find a spurious model, then we add a lemma to refine the abstraction. This is in the spirit of the counter-example-guided abstraction refinement approach [27].

We consider the case where the quantifier-free extensional theory of arrays $T_A$ is combined with a decidable quantifier-free first-order theory $T_B$, e.g. bit-vectors. Our decision procedure uses an abstraction refinement similar to [7, 30, 34]. However, we do not use a propositional skeleton as in [49], but a $T_B$ skeleton as formula abstraction. Similar to STP [35, 36], we replace reads by fresh abstraction variables and iteratively refine the formula abstraction. Therefore, our abstraction refinements are in $T_B$.

This article is an improvement and extension of a preliminary version presented at the SMT'08 workshop [19]. We discuss our decision procedure in a more generic context and prove soundness and completeness. As implementing a decision procedure presented at an abstract and theoretical level is nontrivial, we also provide implementation details and optimizations, in particular in combination with bit-vectors.

This article is organized as follows. In §2 and §3 we provide the necessary theoretical background. We introduce the extensional and non-extensional theory of arrays, and discuss preliminaries for our decision procedure. In §4 we treat preprocessing and in §5 formula abstraction. In §6 we present a high-level overview of our decision procedure. In §7 we informally introduce the main parts of our decision procedure: consistency checking and lemma generation. In §8 we formally define how lemmas are generated and prove soundness. In §9 we prove completeness and in §10 we discuss complexity. In §11 we discuss implementation details and optimizations, and report on experimental results in §12. In §13 we discuss related work, and finally conclude in §14.

## 2. Theory of Arrays

In principle, the theories of arrays are either extensional or non-extensional. While the extensional theories allow to reason about array elements and *arrays*, the non-extensional theories support reasoning about array elements only. McCarthy introduced the classic non-extensional theory of arrays with the help of read-over-write semantics in [44].

A first-order theory is typically defined by its signature and set of axioms. The theory of arrays has the signature $\Sigma_A : \{read, write, =\}$. The function $read(a, i)$ returns the value of array $a$ at index $i$. Arrays are represented in a functional way. The function $write(a, i, e)$ returns array $a$, overwritten at index $i$ with element $e$. All other elements of $a$ remain the same. The predicate $=$ is only applied to array *elements*. The axioms of the theory of arrays are the following:

$$\begin{array}{lrcl} \text{(A1)} & i = j & \Rightarrow & read(a, i) = read(a, j) \\ \text{(A2)} & i = j & \Rightarrow & read(write(a, i, e), j) = e \\ \text{(A3)} & i \neq j & \Rightarrow & read(write(a, i, e), j) = read(a, j) \end{array}$$

Additionally, we assume that the theory of arrays includes the axioms of reflexivity, symmetry and transitivity of equality. If we also want to support equality on arrays, then we need an additional axiom of extensionality:

$$\text{(A4)} \quad a = b \quad \Leftrightarrow \quad \forall i \, (read(a, i) = read(b, i))$$

Note that (A1) and the implication from left to right of (A4) are instances of the function congruence axiom schema. Alternatively, (A4) can be expressed in the following way:

$$\text{(A4}') \quad a \neq b \quad \Leftrightarrow \quad \exists \lambda \, (read(a, \lambda) \neq read(b, \lambda))$$

One can interpret $(A4')$ in the way that if and only if $a$ is unequal to $b$, we have to find a witness of inequality, i.e. we have to find an index $\lambda$ at which the arrays differ.

In the rest of this article we write $T_A$ to denote the quantifier-free fragment of the *extensional* first-order theory of arrays. We write $T_A \models \phi$ to denote that a $\Sigma_A$ formula $\phi$ is valid in $T_A$ ($T_A$-valid). A $\Sigma_A$-formula is $T_A$-valid if every interpretation that satisfies the axioms of $T_A$ also satisfies $\phi$.

## 3. Preliminaries

We assume that we have a decidable quantifier-free first-order theory $T_B$ supporting equality. $T_B$ is defined by its set of axioms and by its signature $\Sigma_B$. We assume that $\Sigma_A \cap \Sigma_B = \{=\}$, i.e. the signatures are disjoint, only equality is shared. Terms are of sort *Base*. Furthermore, we assume that we have a sound and complete decision procedure $DP_B$ such that:

1. $DP_B$ takes a $\Sigma_B$-formula and computes satisfiability modulo $T_B$.
2. If a $\Sigma_B$-formula is satisfiable, $DP_B$ returns a $B$-model $\sigma$ mapping terms and formulas to concrete values.

In the literature, models are satisfying assignments to variables only. However, we assume that $DP_B$ also provides consistent assignments to arbitrary terms and formulas in the input formula. Implementing this feature is typically straightforward as we can replace variables with their concrete assignments and recursively evaluate terms and formulas to obtain each assignment. If we use a SAT solver inside $DP_B$ in combination with some variant of Tseitin encoding [52], then we can directly evaluate the assignments to the auxiliary Tseitin variables of each term resp. formula.

Our decision procedure $DP_A$ decides satisfiability modulo $T_A \cup T_B$. The signature of $T_A$ is augmented with the signature of $T_B$ and the combined theory $T_A \cup T_B$ includes the axioms of $T_A$ and $T_B$. Array variables and writes are terms of sort *Array*. They have sets of indices and values of sort *Base*[1]. Our decision procedure takes a $(\Sigma_A \cup \Sigma_B)$-formula $\phi$ as input and decides satisfiability. If it is satisfiable, then our decision procedure provides an $A$-model. In contrast to $B$-models, $A$-models additionally provide concrete assignments to terms of sort *Array*.

## 4. Preprocessing

We apply the following two preprocessing steps to the input formula $\phi$. We assume that $\phi$ is represented as Directed Acyclic Graph (DAG) with structural hashing enabled, i.e. syntactically identical subformulas and subterms are shared. Furthermore, we assume that inequality is represented as combination of equality and negation, i.e. $a \neq b$ is represented as $\neg(a = b)$.

1. For each equality $a = c \in \phi$ between terms of sort *Array*, we introduce a fresh variable $\lambda$ of sort *Base* and two *virtual* reads $read(a, \lambda)$ and $read(c, \lambda)$. Then, we add the following top-level constraint:
$$a \neq c \quad \rightarrow \quad \exists \lambda.\ read(a, \lambda) \neq read(c, \lambda)$$
2. For each $write(a, i, e) \in \phi$, we add the following top-level constraint:
$$read(write(a, i, e), i) = e$$

These steps add additional top-level constraints $c_1 \ldots c_n$ to $\phi$ to the resulting formula $\pi$. To be more precise, $\pi$ is defined as follows:

$$\pi := \phi \ \wedge \bigwedge_{i=1}^{n} c_i$$

The idea of preprocessing step 1 is that virtual reads are used as witness for array *inequality*. If $a \neq b$, then it must be possible to find an index $\lambda$ at which the arrays contain different values. A similar usage of $\lambda$ can be found in [16], and as $k$ in rule *ext* [51].

The idea of preprocessing step 2 is that additional reads are introduced to enforce consistency on write values. This preprocessing step simplifies our presentation and proofs as we can focus on reads and do not have to explicitly treat write values. In contrast to preprocessing step 1, we do not expect that step 2 is performed in real implementations. In section 11.4 we discuss how this preprocessing step can be avoided.

**Proposition 4.1.** *$\phi$ and $\pi$ are equisatisfiable.*

*Proof.* We show that each constraint $c$ added by a preprocessing step is $T_A$-valid: $T_A \models c$. Therefore, conjoining these constraints to $\phi$ does not affect satisfiability.

Let $c$ be an instance of preprocessing step 1. Axiom $(A4')$ asserts $c$. Thus, $T_A \models c$. Let $c$ be an instance of preprocessing step 2. Axiom $(A2)$ asserts $c$. Thus, $T_A \models c$. □

1. In principle, the sort of indices may differ from the sort of values. However, like most decision procedure descriptions, e.g. [51], we assume that the sorts are the same to simplify presentation and proofs. It is straightforward to generalize our decision procedure to support multiple sorts.

### 4.1 If-Then-Else on Arrays

Typically, modern SMT solvers support an if-then-else on terms of sort *Array*: $cond(c, a, b)$, where $c$ is a boolean condition. We do not explicitly treat this feature here to simplify our presentation. In principle, $cond(c, a, b)$ can always be replaced by a fresh variable $d$ of sort *Array*, and the following constraint:

$$c \;\rightarrow\; d = a \quad \wedge \quad \neg c \;\rightarrow\; d = b$$

This preprocessing step must be performed before preprocessing step 1 and 2. In principle, our approach supports a direct integration of if-then-else on terms of sort *Array* without rewriting it up front [19].

## 5. Formula Abstraction

In the following, we consider a partial formula abstraction function $\alpha$. Our formula abstraction is similar to the abstraction in the lazy SMT approach [49], but we do *not* generate a pure propositional skeleton, but a $T_B$ skeleton as formula abstraction. Similar to STP [35, 36], we replace reads by fresh variables. To be precise, our approach introduces abstraction variables of sort *Base*, and propositional abstraction variables to handle extensionality. The formula abstraction is applied after preprocessing.

Let $\pi$ be the result of preprocessing a $(\Sigma_A \cup \Sigma_B)$-formula $\phi$. Analogously to $\phi$, we assume that $\pi$ is represented as DAG with structural hashing enabled. We also assume that inequality is represented as combination of equality and negation, i.e. $a \neq b$ is represented as $\neg(a = b)$. The abstraction function $\alpha$ recurses down the structure of $\pi$ and builds the over-approximation $\alpha(\pi)$.

For terms of sort *Array* the result of applying $\alpha$ is undefined. The abstraction $\alpha$ maps:

1. Each read $read(a, i)$ to a fresh abstraction variable of sort *Base*.
2. Each equality $a = b$ between terms of sort *Array* to a fresh propositional abstraction variable.
3. Each term $f(t_1, \ldots, t_m)$ of sort *Base* and each formula to $f(\alpha(t_1), \ldots, \alpha(t_m))$.
4. Each (non-array) variable and symbolic constant to itself.

Now, assume that we start from the boolean part of $\pi$, then terms of sort *Array* can only be reached by passing reads, and equalities between terms of sort *Array*. The abstraction function $\alpha$ replaces exactly these terms by fresh variables, hence the underlying terms of sort *Array* are no longer reachable in $\alpha(\pi)$.

**Proposition 5.1.** *$\alpha(\pi)$ is an over-approximating abstraction of $\pi$.*

*Proof.* First, we show that whenever we have a model $\sigma$ for $\pi$, then we can construct a model $\sigma_\alpha$ for $\alpha(\pi)$. We assume that $\sigma$ not only returns satisfying assignments to variables, but also consistent assignments to terms and formulas in $\pi$. Given $\sigma$, $\sigma_\alpha$ can be constructed as follows: For each read $r$ in $\pi$ define $\sigma_\alpha(\alpha(r)) := \sigma(r)$. For each equality $e$ between terms of sort *Array* in $\pi$ define $\sigma_\alpha(\alpha(e)) := \sigma(e)$. For all other terms and formulas $x$ in $\alpha(\pi)$,

define $\sigma_\alpha(x) := \sigma(x)$. Obviously, $\sigma_\alpha$ satisfies $\alpha(\pi)$ as the abstraction variables are fresh and unconstrained.

Second, we show that that whenever we have a model $\sigma_\alpha$ for $\alpha(\pi)$, we can *not* always construct a model $\sigma$ for $\pi$. Consider the following example. Assume that our theory $T_B$ is the quantifier-free theory of Presburger arithmetic, i.e. *Base* represents the natural numbers. Furthermore, assume that $i$, $\lambda$ and $e$ are terms of sort *Base*, and $a$ is an array with indices and values of sort *Base*. Let $w$ be $write(a, i, e)$. Consider the following formula $\pi_1$:

$$w = a \ \wedge \ read(a, i) \neq e \ \wedge \ (w \neq a \ \rightarrow \ read(w, \lambda) \neq read(a, \lambda)) \ \wedge \ read(w, i) = e$$

Let $q$ be $\alpha(w = a)$, $s$ be $\alpha(read(a, i))$, $t$ be $\alpha(read(w, \lambda))$, $u$ be $\alpha(read(a, \lambda))$, and $v$ be $\alpha(read(w, i))$. The abstraction $\alpha(\pi_1)$ results in the following formula:

$$q \ \wedge \ s \neq e \ \wedge \ (\neg q \ \rightarrow \ t \neq u) \ \wedge \ v = e$$

Obviously, we can find a model $\sigma_\alpha$. However, $\pi_1$ is unsatisfiable as $w = a$, $read(a, i) \neq e$, and $read(w, i) = e$ can not all be assigned to $\top$. □

As $\alpha(\pi)$ is an over-approximating abstraction of $\pi$, $\alpha(\pi)$ may have additional models that are spurious in $\pi$. Therefore, we have to find a way to eliminate spurious models, which we will discuss in the next sections.

## 6. Decision Procedure

Our decision procedure $DP_A$ uses an abstraction refinement loop similar to [7, 30, 34]. However, we use a $T_B$ skeleton as formula abstraction. Our formula abstraction introduces abstraction variables of sort *Base*. Boolean abstraction variables are only introduced to handle extensionality.

In our decision procedure the over-approximated formula is solved by the underlying decision procedure $DP_B$. If we find a spurious model, then we add a lemma to refine the abstraction. In each iteration the algorithm may terminate concluding unsatisfiability, or satisfiability if the model is not spurious. However, if the model is spurious, inconsistent assignments are ruled out, and the procedure continues with the next iteration. An overview of $DP_A$ is shown in Fig. 1.

```
procedure DP_A (φ)
    π ← preprocess (φ)
    ξ ← ⊤
    loop
        (r, σ) ← DP_B (α(π) ∧ ξ)
        if (r = unsatisfiable) return unsatisfiable
        if (consistent (π, σ)) return satisfiable
        ξ ← ξ ∧ α(lemma(π, σ))
```

First, we preprocess $\phi$ and initialize our formula refinement $\xi$ to $\top$. In each loop iteration, we take the abstraction $\alpha(\pi)$, conjoin it with our formula refinement, and run our decision

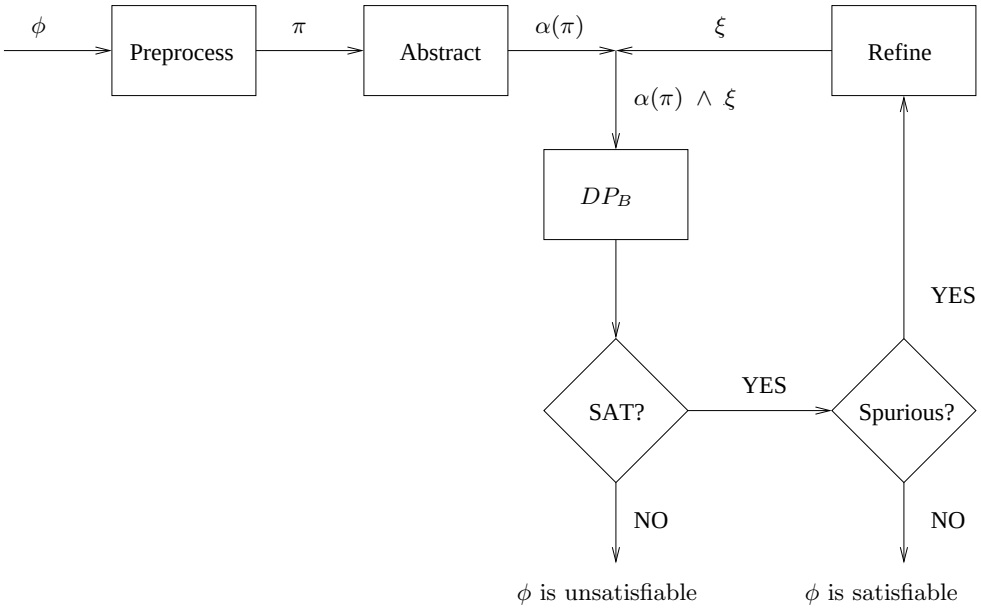


**Figure 1.** Overview of $DP_A$.

procedure $DP_B$. If $DP_B$ concludes that $\alpha(\pi)$ is unsatisfiable, then we can conclude that $\pi$ is unsatisfiable as $\alpha$ is an over-approximating abstraction. As $\phi$ and $\pi$ are equisatisfiable, we can finally conclude that the original formula $\phi$ is unsatisfiable.

However, if $DP_B$ concludes that $\alpha(\pi) \wedge \xi$ is satisfiable, it returns a satisfying assignment $\sigma$, i.e. a $B$-model, which can be used to build an $A$-model of $\pi$. As we have used an over-approximating abstraction, we have to check if this is a spurious model. We run our consistency checker on the preprocessed formula $\pi$. The consistency checker checks whether the $B$-model can be extended to a valid $A$-model or not. If the consistency checker does not find a conflict, then we can conclude that $\pi$ is satisfiable. Again, as $\pi$ and $\phi$ are equisatisfiable, we can finally conclude that $\phi$ is satisfiable. However, if the current assignment $\sigma$ is invalid with respect to the extensional theory of arrays, then we generate a lemma as formula refinement and continue with the next iteration.

## 7. Consistency Checking and Lemma Generation

In the following, we discuss the consistency checking algorithm, denoted by *consistent* in $DP_A$, and lemma generation, denoted by *lemma* in $DP_A$. A more precise description of how the lemmas are generated is part of the soundness proof in section 8.

The consistency checker takes $\pi$ and a concrete $B$-model $\sigma$ for $\alpha(\pi)$ and checks whether it can be extended to a valid $A$-model or not. In principle, the algorithm is based on read propagation and congruence checking of reads. In the first phase, the consistency checker propagates reads to other terms of sort *Array*. After the propagation has finished, the consistency checker iterates over all terms of sort *Array* in $\pi$ and checks whether the reads are congruent or not. If they are not, a lemma is generated to refine the abstraction.

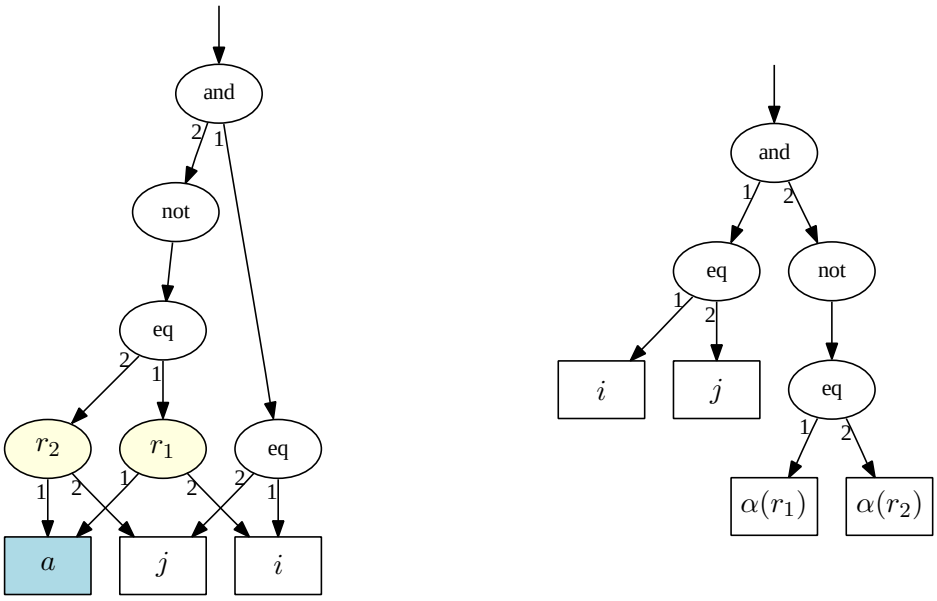


**Figure 2.** Example 1. Formula $\phi$ (resp. $\pi$) with array $a$ and two reads $r_1$ and $r_2$ is shown left. The label *and* means conjunction, *not* means negation, and *eq* means equality. Edge numbers denote the ordering of the operands, e.g. $r_1$ is a read function where the first operand is $a$, and the second operand is $i$. The abstraction $\alpha(\pi)$ is shown right.

### 7.1 Reads

First, we consider the case where reads may only be applied to array variables. Moreover, we assume that we have no writes and also no equalities between arrays. In this case no read propagation is necessary. The consistency checker iterates over all array variables and checks if congruence on reads is violated. Axiom (A1) is violated if and only if:

$$\sigma(\alpha(i)) = \sigma(\alpha(j)) \ \wedge \ \sigma(\alpha(read(a,i))) \neq \sigma(\alpha(read(a,j)))$$

In this case we generate the following lemma:

$$i = j \quad \Rightarrow \quad read(a,i) = read(a,j)$$

In principle, this is just a lazy variant of Ackermann expansion [3]. In particular, Ackermann expansion is an interesting topic for SMT as it plays an important role in deciding formulas in the theory of equality and uninterpreted functions. For example, see [23].

**Example 1.**

*Let $\phi$ be as shown in Fig. 2. The preprocessed formula $\pi$ is identical to $\phi$ as no preprocessing steps are necessary.*

*We consider the set of reads $\{r_1, r_2\}$. We run $DP_B$ on $\alpha(\pi)$ and $DP_B$ generates a model $\sigma$ such that $\sigma(i) = \sigma(j)$ and $\sigma(\alpha(r_1)) \neq \sigma(\alpha(r_2))$. We find an inconsistency as $\sigma(i) = \sigma(j)$, but $\sigma(\alpha(r_1)) \neq \sigma(\alpha(r_2))$, and generate the following lemma:*

$$i = j \quad \Rightarrow \quad r_1 = r_2$$

Note that in this example $\alpha(i) = i$ and $\alpha(j) = j$, but this is not the general case. Read values may also be used as read indices. Furthermore, note that our generated lemmas are in $T_A$. However, in order to refine our abstraction $\alpha(\pi)$, we apply the abstraction function $\alpha$ to each generated lemma which is not shown in our examples.

### 7.2 Writes

If we additionally consider writes in $\phi$ resp. $\pi$, then a term of sort *Array* is either an array variable or a write. Note that it is possible to *nest* writes. We introduce a mapping $\rho$ which maps terms of sort *Array* to a set of reads. Note that these reads are drawn from those appearing in $\pi$. In the first phase, the consistency checker initializes $\rho$ for each term of sort *Array*. Then, it iterates over all writes and propagates reads if necessary:

*I*. Map each $b$ to its set of reads $\rho(b)$, initialized as $\rho(b) = \{read(b, i)$ in $\pi$ for some $i\}$.

*D*. For each $read(b, i) \in \rho(write(a, j, e))$: if $\sigma(\alpha(i)) \neq \sigma(\alpha(j))$, add $read(b, i)$ to $\rho(a)$.

Repeat rule *D* until fix-point is reached, i.e. $\rho$ does not change anymore. Fix-point computation of $\rho$ can be implemented as post-order traversal on the array expression subgraph starting at "top-level" writes, i.e. writes that are not overwritten by other writes. Note that read propagation in rule *D* is the sense of copying without removing, i.e if we propagate a read $r$ from source $s$ to destination $d$ it actually means that we copy $r$ from $\rho(s)$ to $\rho(d)$.

In the second phase we check congruence:

*C*. For each pair $read(b, i)$, $read(c, k)$ in $\rho(a)$: check *adapted* congruence axiom.

In rule *C* the original array congruence axiom (A1) can not be applied as $\rho(a)$ may contain propagated reads that do not read on $a$ directly. Therefore, we have to adapt (A1). The adapted axiom is violated if and only if:

$$\sigma(\alpha(i)) = \sigma(\alpha(k)) \ \wedge \ \sigma(\alpha(read(b, i))) \neq \sigma(\alpha(read(c, k)))$$

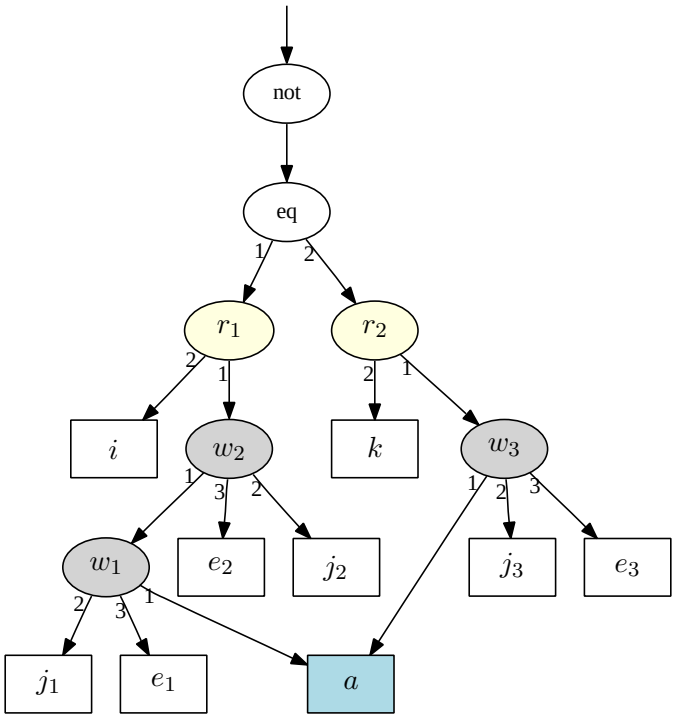


**Figure 3.** Formula $\phi$ for examples 2 and 3. It has one array variable $a$, two reads $r_1$ and $r_2$, and three writes $w_1$ to $w_3$.

We collect all indices $j_1^i \ldots j_m^i$ that have been used as $j$ in $D$ while propagating $read(b, i)$. Analogously, we collect all indices $j_1^k \ldots j_n^k$ that have been used as $j$ in $D$ while propagating $read(c, k)$. Then, we generate the following lemma:

$$i = k \ \wedge \ \bigwedge_{l=1}^{m} i \neq j_l^i \ \wedge \ \bigwedge_{l=1}^{n} k \neq j_l^k \quad \Rightarrow \quad read(b, i) = read(c, k)$$

The first big conjunction represents that $\sigma(\alpha(i))$ is different from all the assignments to the write indices on the propagation path of $read(b, i)$. Analogously, the second conjunction represents that $\sigma(\alpha(k))$ is different from all the assignments to the write indices on the propagation path of $read(c, k)$. The resulting lemma ensures that the current inconsistency can not occur anymore in future refinement iterations. Either the propagation paths change or the reads are congruent.

In this section we keep the description of our lemmas rather informal. A more precise description is part of the soundness proof in section 8.

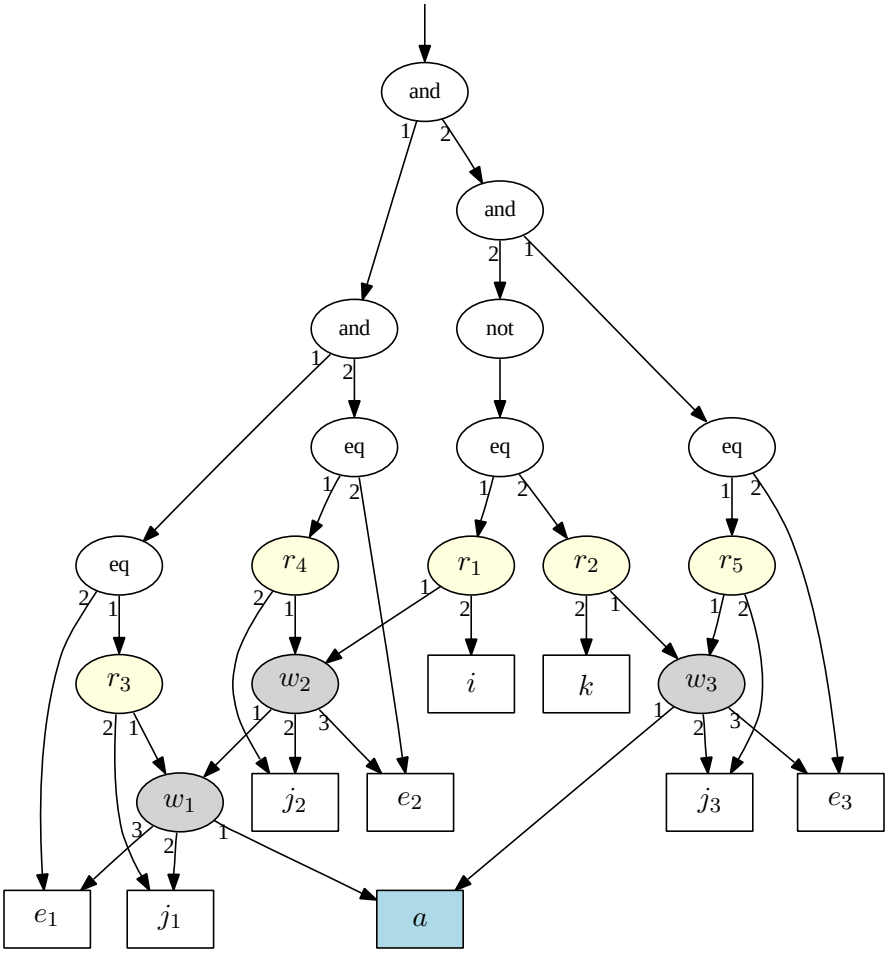


**Figure 4.** Formula $\pi$ for examples 2 and 3. Reads $r_3$ to $r_5$ have been added by preproc. step 2.

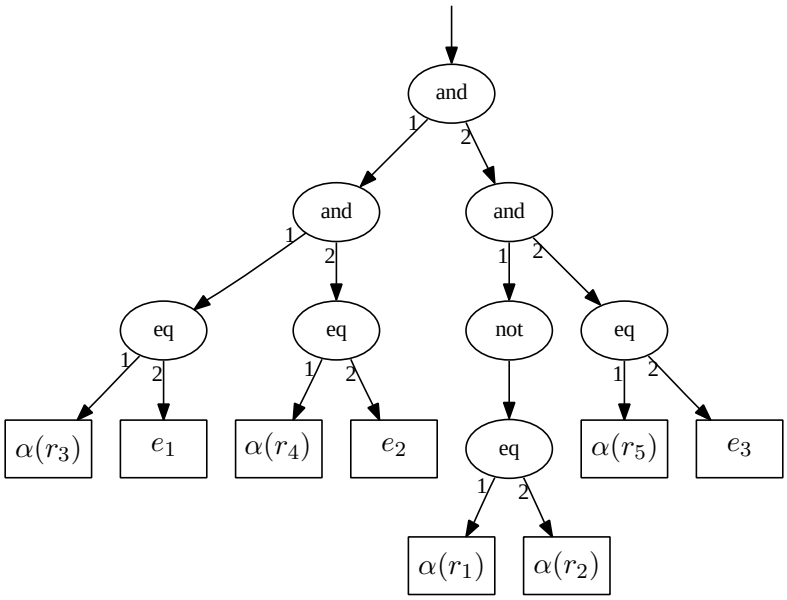


**Figure 5.** Formula $\alpha(\pi)$ for examples 2 and 3. The read indices $i$ and $k$, and the write indices $j_1$ to $j_3$ are not shown as they are not reachable from the root of $\alpha(\pi)$. Initially, they are unconstrained and $DP_B$ can assign them arbitrarily.

**Example 2.**

*Let $\phi$ be as shown in Fig. 3. The preprocessed formula $\pi$ is shown in Fig. 4, and $\alpha(\pi)$ is shown in Fig. 5.*

*We run $DP_B$ on $\alpha(\pi)$ and assume that it generates a model $\sigma$ such that $\sigma(i) = \sigma(k)$, $\sigma(i) \neq \sigma(j_2)$, $\sigma(i) \neq \sigma(j_1)$, $\sigma(k) \neq \sigma(j_3)$, i.e. the assignments to the write indices are different from the assignments to the read indices $i$ and $k$. Furthermore, $\sigma(\alpha(r_1)) \neq \sigma(\alpha(r_2))$. Note that if $DP_B$ finds a model, then $\sigma(\alpha(r_3)) = \sigma(e_1)$, $\sigma(\alpha(r_4)) = \sigma(e_2)$, $\sigma(\alpha(r_5)) = \sigma(e_3)$, and $\sigma(\alpha(r_1)) \neq \sigma(\alpha(r_2))$ has to hold.*

*Initially, $\rho(a) = \emptyset$, $\rho(w_1) = \{r_3\}$, $\rho(w_2) = \{r_1, r_4\}$, and $\rho(w_3) = \{r_2, r_5\}$, Read $r_1$ is propagated down to $a$ as $\sigma(i) \neq \sigma(j_2)$ and $\sigma(i) \neq \sigma(j_1)$. Analogously, read $r_2$ is propagated down to $a$ as $\sigma(k) \neq \sigma(j_3)$. Therefore, $\rho(a) = \{r_1, r_2\}$, $\rho(w_1) = \{r_1, r_3\}$, $\rho(w_2\} = \{r_1, r_4\}$, and $\rho(w_3) = \{r_2, r_5\}$, We check $\rho(a)$, find an inconsistency according to rule $C$ as $\sigma(i) = \sigma(k)$, but $\sigma(\alpha(r_1)) \neq \sigma(\alpha(r_2))$, and generate the following lemma:*

$$i = k \ \wedge \ i \neq j_2 \ \wedge \ i \neq j_1 \ \wedge \ k \neq j_3 \quad \Rightarrow \quad r_1 = r_2$$

**Example 3.**

*Again, let $\phi$ be as shown in Fig. 3. The preprocessed formula $\pi$ is shown in Fig. 4, and $\alpha(\pi)$ is shown in Fig. 5.*

*We run $DP_B$ on $\alpha(\pi)$ and assume that $DP_B$ generates a model $\sigma$ such that $\sigma(i) \neq \sigma(j_2)$, $\sigma(i) = \sigma(j_1)$, $\sigma(\alpha(r_1)) \neq \sigma(e_1)$, $\sigma(k) = \sigma(j_3)$, and $\sigma(\alpha(r_2)) = \sigma(\alpha(r_5))$. Again, note that $\sigma(\alpha(r_3)) = \sigma(e_1)$, $\sigma(\alpha(r_4)) = \sigma(e_2)$, $\sigma(\alpha(r_5)) = \sigma(e_3)$, and $\sigma(\alpha(r_1)) \neq \sigma(\alpha(r_2))$.*

*Initially, $\rho(a) = \emptyset$, $\rho(w_1) = \{r_3\}$, $\rho(w_2) = \{r_1, r_4\}$, and $\rho(w_3) = \{r_2, r_5\}$, We propagate $r_1$ down to $w_1$ as $\sigma(i) \neq \sigma(j_2)$. Therefore, we update $\rho(w_1)$ to $\{r_1, r_3\}$. We check $\rho(w_1)$, find an inconsistency according to rule $C$ as $\sigma(i) = \sigma(j_1)$, but $\sigma(\alpha(r_1)) \neq \sigma(\alpha(r_3))$, and generate the following lemma:*

$$i = j_1 \ \wedge \ i \neq j_2 \quad \Rightarrow \quad r_1 = r_3$$

### 7.3 Equalities between Arrays

We also consider equalities between terms of sort *Array*. In particular, we add rules $R$ and $L$ to propagate reads over equalities between terms of sort *Array*. Furthermore, we add rule $U$ to propagate reads upwards if they are not overwritten.

$U$. For each $read(b, i) \in \rho(a)$: if $\sigma(i) \neq \sigma(j)$, add $read(b, i)$ to $\rho(write(a, j, e))$.

$R$. For each $a = c$, $\sigma(\alpha(a = c)) = \top$: for each $read(b, i) \in \rho(a)$: add $read(b, i)$ to $\rho(c)$.

$L$. For each $a = c$, $\sigma(\alpha(a = c)) = \top$: for each $read(b, i) \in \rho(c)$: add $read(b, i)$ to $\rho(a)$.

Rules $R$ and $L$ represent that we also have to propagate reads over equalities between terms of sort *Array* to ensure extensional consistency, but only if $DP_B$ assigns them to $\top$. Rule $U$ is responsible to propagate reads upwards, but only if the assignment to the write index is different from the assignment to the read index. Note, $write(a, j, e)$ must appear in $\pi$.

In order to implement our consistency checking algorithm, we need a real fix-point computation for $\rho$. This can be implemented by a working queue that manages future read propagations. Simple post-order traversal on the array expression subgraph of $\pi$ is no longer sufficient as reads are not only propagated downwards, but also upwards, and between equalities on terms of sort *Array*.

As soon as we consider extensionality, propagating reads upwards is necessary. Consider the following example, which is also shown in Fig. 10:

$$write(a, i, e_1) = write(b, j, e_2) \ \wedge \ i \neq k \ \wedge \ j \neq k \ \wedge \ read(a, k) \neq read(b, k)$$

The write indices $i$ and $j$ are respectively different from the read index $k$. Therefore, position $k$ at array $a$ is not overwritten with $e_1$. Analogously, position $k$ at array $b$ is not overwritten with $e_2$. However, in combination with $i \neq k$ and $j \neq k$, the equality between the two writes

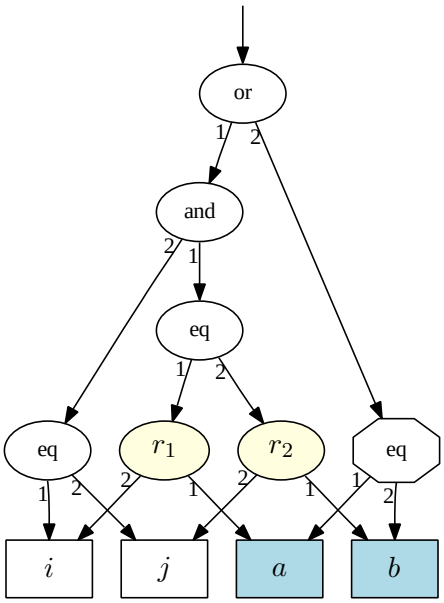


**Figure 6.** Extensional formula $\phi$ for example 4.

enforces that the two reads have to be extensionally congruent. Therefore, this formula is unsatisfiable.

In order to detect extensionally inconsistent reads, it is necessary to propagate them to the same term of sort *Array*. Rule $U$ enforces extensional consistency in combination with rules $L$ and $R$. Reads at array positions that are not overwritten are propagated upwards to writes, and between equalities on terms of sort *Array*. In this way, extensionally inconsistent reads are propagated to the same term of sort *Array*, and rule $C$ can detect the inconsistency. This is demonstrated in example 5. Note that upward propagation respects (A3). We propagate reads upwards only if the assignments to the read indices differ from the assignment to the write indices, i.e. the values are not overwritten.

Lemmas generated upon inconsistency detection are extended as follows. Lemmas involving rule $U$ are extended analogously to rule $D$. Lemmas involving rules $L$ and $R$ are extended in the following way. We additionally collect all array equalities $x_1^i, \ldots x_q^i$ used in rule $R$ or $L$ while propagating $read(b, i)$. Analogously, we additionally collect all array equalities $x_1^k, \ldots x_r^k$ used in rule $R$ or $L$ while propagating $read(c, k)$. The lemma is extended in the following way:

$$\ldots \wedge \bigwedge_{l=1}^{q} x_l^i \wedge \bigwedge_{l=1}^{r} x_l^k \quad \Rightarrow \quad read(b, i) = read(c, k)$$

The complete set of our rules implementing our consistency checking algorithm based on read propagation is summarized in Fig. 9.

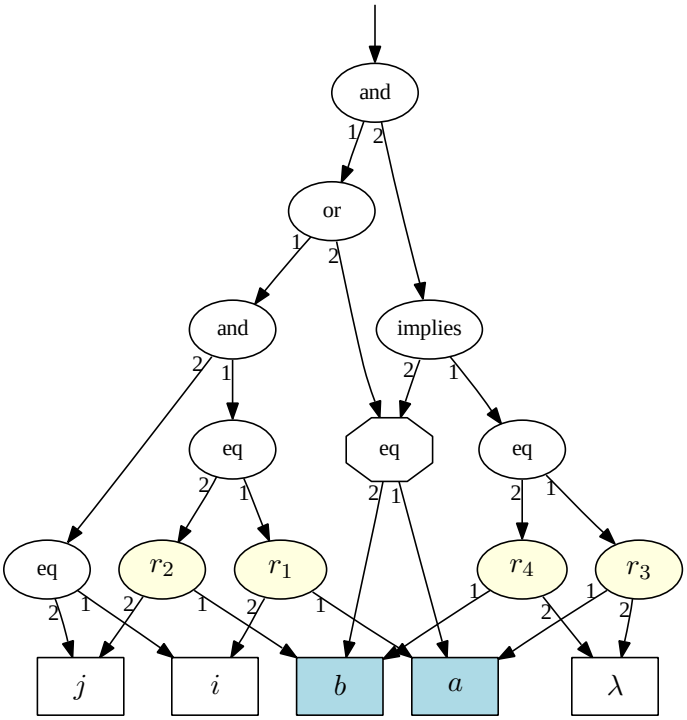


**Figure 7.** Formula $\pi$ for example 4. The right part of the formula has been added by preprocessing step 1. Here we use $r_3 = r_4 \rightarrow a = b$, which is obviously equal to $a \neq b \rightarrow r_3 \neq r_4$. Note that $r_3$ and $r_4$ are virtual reads that do not occur in the original formula $\phi$.

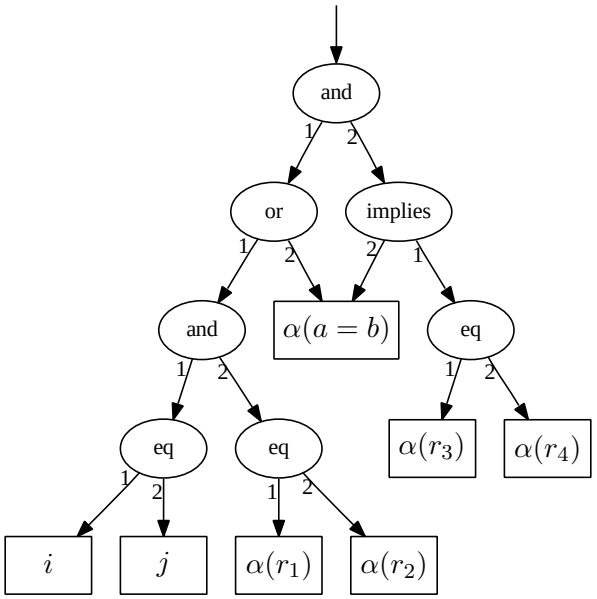


**Figure 8.** Formula $\alpha(\pi)$ for example 4.

$I$. Map each $b$ to its set of reads $\rho(b)$, initialized as $\rho(b) = \{read(b, i) \text{ in } \pi\}$.

$D$. For each $read(b, i) \in \rho(write(a, j, e))$: if $\sigma(\alpha(i)) \neq \sigma(\alpha(j))$, add $read(b, i)$ to $\rho(a)$.

$U$. For each $read(b, i) \in \rho(a)$: if $\sigma(\alpha(i)) \neq \sigma(\alpha(j))$, add $read(b, i)$ to $\rho(write(a, j, e))$.

$R$. For each $a = c$, $\sigma(\alpha(a = c)) = \top$: for each $read(b, i) \in \rho(a)$: add $read(b, i)$ to $\rho(c)$.

$L$. For each $a = c$, $\sigma(\alpha(a = c)) = \top$: for each $read(b, i) \in \rho(c)$: add $read(b, i)$ to $\rho(a)$.

$C$. For each pair $read(b, i)$, $read(c, k)$ in $\rho(a)$: check *adapted* congruence axiom.

**Figure 9.** Final consistency checking algorithm. Rule $I$ initializes $\rho$. For each term $b$ of sort *Array*, i.e. array variables and writes, $\rho(b)$ is initialized as set of reads that directly read on $b$. Rules $D$ resp. $U$ perform downward resp. upward propagation. Rules $L$ resp. $R$ perform left resp. right propagation over equalities between terms of sort *Array*. Rules $D$, $U$, $R$ and $L$ are repeated until fix-point is reached, i.e. $\rho$ does not change anymore. However, in principle, consistency checking rule $C$ may also be interleaved with $D$, $U$, $R$ and $L$. This means that consistency checking is performed on-the-fly.

**Example 4.**

*Let $\phi$ be as in Fig. 6. The preprocessed formula $\pi$ is shown in Fig. 7, and $\alpha(\pi)$ is shown in Fig. 8.*

*We run $DP_B$ on $\alpha(\pi)$ and assume that $DP_B$ generates a model $\sigma$ such that $\sigma(i) = \sigma(j)$, $\sigma(\alpha(a = b)) = \top$, and $\sigma(\alpha(r_1)) \neq \sigma(\alpha(r_2))$.*

*Initially, $\rho(a) = \{r_1, r_3\}$ and $\rho(b) = \{r_2, r_4\}$. We propagate $r_1$ and $r_3$ from $a$ to $b$ as $\sigma(\alpha(a = b)) = \top$ and therefore extensional consistency has to be enforced. Analogously, we propagate $r_2$ and $r_4$ from $b$ to $a$. Therefore, $\rho(a) = \rho(b) = \{r_1, r_2, r_3, r_4\}$. We check $\rho(a)$, find an inconsistency according to rule $C$ as $\sigma(i) = \sigma(j)$, but $\sigma(\alpha(r_1)) \neq \sigma(\alpha(r_2))$. We generate the following lemma:*

$$i = j \ \wedge \ a = b \quad \Rightarrow \quad r_1 = r_2$$

**Example 5.**

*Let $\phi$ be as shown in Fig. 10. The preprocessed formula $\pi$ is shown in Fig. 11, and $\alpha(\pi)$ is shown in Fig. 12.*

*We run $DP_B$ on $\alpha(\pi)$ and assume that $DP_B$ generates a model $\sigma$ such that $\sigma(k) \neq \sigma(i)$, $\sigma(k) \neq \sigma(j)$, $\sigma(\alpha(r_1)) \neq \sigma(\alpha(r_2))$, and $\sigma(\alpha(w_1 = w_2)) = \top$. Furthermore, $\sigma(\alpha(r_5)) = \sigma(e_1)$, and $\sigma(\alpha(r_6)) = \sigma(e_2)$.*

*We perform on-the-fly consistency checking in depth-first search manner. Initially, $\rho(a) = \{r_1\}$, $\rho(b) = \{r_2\}$, $\rho(w_1) = \{r_3, r_5\}$, and $\rho(w_2) = \{r_4, r_6\}$. We propagate $r_1$ up to $w_1$ as $\sigma(k) \neq \sigma(i)$, i.e. the value has not been overwritten by $w_1$. We update $\rho(w_1)$ to $\{r_1, r_3, r_5\}$. With respect to $\sigma$, we assume that the reads in $\rho(w_1)$ do not violate congruence, i.e on-the-fly consistency checking rule C does not find a conflict in $\rho(w_1)$. To enforce extensional consistency, we propagate $r_1$ from $w_1$ to $w_2$ as $\sigma(\alpha(w_1 = w_2)) = \top$. We update $\rho(w_2)$ to $\{r_1, r_4, r_6\}$. Again, we assume that the reads in $\rho(w_2)$ do not violate congruence. Then, we propagate $r_1$ from $w_2$ down to $b$ as $\sigma(k) \neq \sigma(j)$. We update $\rho(b)$ to $\{r_1, r_2\}$. We check $\rho(b)$, find an inconsistency according to rule C as $r_1$ and $r_2$ read on the same index, but read a different value. We generate the following lemma:*

$$k \neq i \ \wedge \ k \neq j \ \wedge \ w_1 = w_2 \quad \Rightarrow \quad r_1 = r_2$$

**Example 6.**

*Again, let $\phi$ be as shown in Fig. 10. The preprocessed formula $\pi$ is shown in Fig. 11, and $\alpha(\pi)$ is shown in Fig. 12.*

*We run $DP_B$ on $\alpha(\pi)$ and assume that $DP_B$ generates a model $\sigma$ such that $\sigma(i) = \sigma(j)$, $\sigma(e_1) \neq \sigma(e_2)$, and $\sigma(\alpha(w_1 = w_2)) = \top$. Furthermore, $\sigma(r_5) = \sigma(e_1)$, $\sigma(r_6) = \sigma(e_2)$, and $\sigma(\alpha(r_1)) \neq \sigma(\alpha(r_2))$.*

*Again, we perform on-the-fly consistency checking in depth-first search manner. Initially, $\rho(a) = \{r_1\}$, $\rho(b) = \{r_2\}$, $\rho(w_1) = \{r_3, r_5\}$, and $\rho(w_2) = \{r_4, r_6\}$. We propagate $r_5$ from $w_1$ to $w_2$ as $\sigma(\alpha(w_1 = w_2)) = \top$. We update $\rho(w_2)$ to $\{r_4, r_5, r_6\}$. We check $\rho(w_2)$ and find an inconsistency according to rule C as $\sigma(i) = \sigma(j)$, but $\sigma(r_5) \neq \sigma(r_6)$. We generate the following lemma:*

$$i = j \ \wedge \ w_1 = w_2 \quad \Rightarrow \quad r_5 = r_6$$

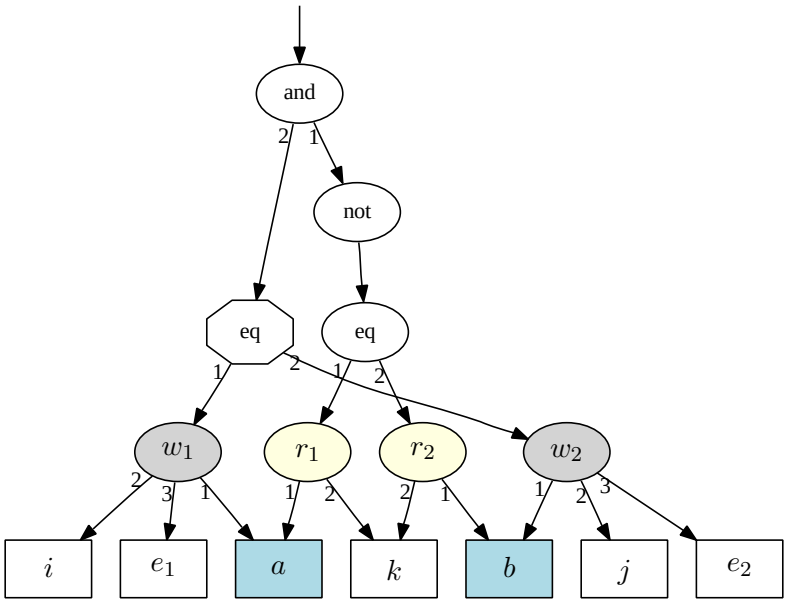


**Figure 10.** Extensional formula $\phi$ for examples 5 and 6.

## 8. Soundness

We show that our approach is sound, i.e. whenever $DP_A$ concludes that $\phi$ is unsatisfiable, then $\phi$ is unsatisfiable modulo $T_A \cup T_B$. In particular, we show that each lemma $l$ generated upon inconsistency detection is $T_A$-valid: $T_A \models l$.

First of all, we introduce a partial mapping $\chi(a, r)$, which maps a term of sort *Array* and a read $r = read(b, i)$ of sort *Base* to a propagation condition $\chi$. If the adapted congruence axiom is violated (rule $C$), then the lemma is obtained by combining propagation conditions of the inconsistent reads. Thus, this section also provides a more formal definition of how lemmas are constructed.

Initially, in rule $I$, $\chi(a, read(a, i)) := \top$ for each read in $\rho(a)$. Otherwise, it is undefined. Whenever we propagate a read $r$ from source $s$ to destination $d$ under the condition $\Delta$, then we set the propagation condition $\chi(d, r)$ to $\chi(s, r) \wedge \Delta$. A propagation occurs only if $\rho(d, r)$ is undefined, i.e. the read has not been propagated to $d$ before. For each rule in Fig. 9, table 1 shows how $\chi$ is updated while propagating reads.

**Lemma 8.1.** $T_A \models \chi(d, read(b, i)) \Rightarrow read(b, i) = read(d, i)$

*Proof.* The proof is by induction over the updates to $\chi$ resp. $\rho$.

$I$**:** trivially holds: $T_A \models \top \Rightarrow read(b, i) = read(b, i)$.

$D$**:** Let $\psi$ be $\chi(write(a, j, e), read(b, i))$ and $\Delta$ be $i \neq j$. From the induction invariant we know $T_A \models \psi \Rightarrow read(write(a, j, e), i) = read(b, i)$. From (A2) we obtain $\Delta \Rightarrow read(write(a, j, e), i) = read(a, i)$. Therefore, we can conclude $T_A \models \psi \wedge \Delta \Rightarrow read(a, i) = read(write(a, j, e), i) = read(b, i)$.

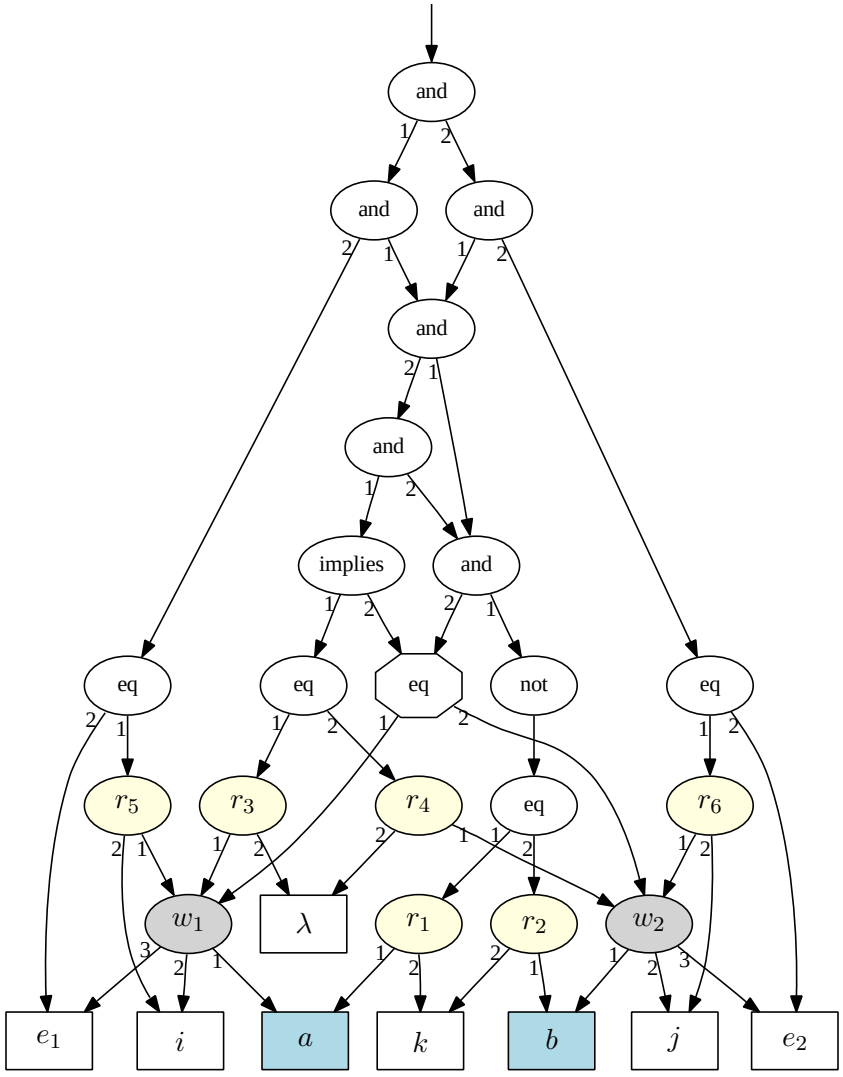


**Figure 11.** Formula $\pi$ for examples 5 and 6. The two virtual reads $r_3$ and $r_4$ have been introduced by preprocessing step 1. Reads $r_5$ and $r_6$ have been introduced by preprocessing step 2 to enforce consistency on write values.

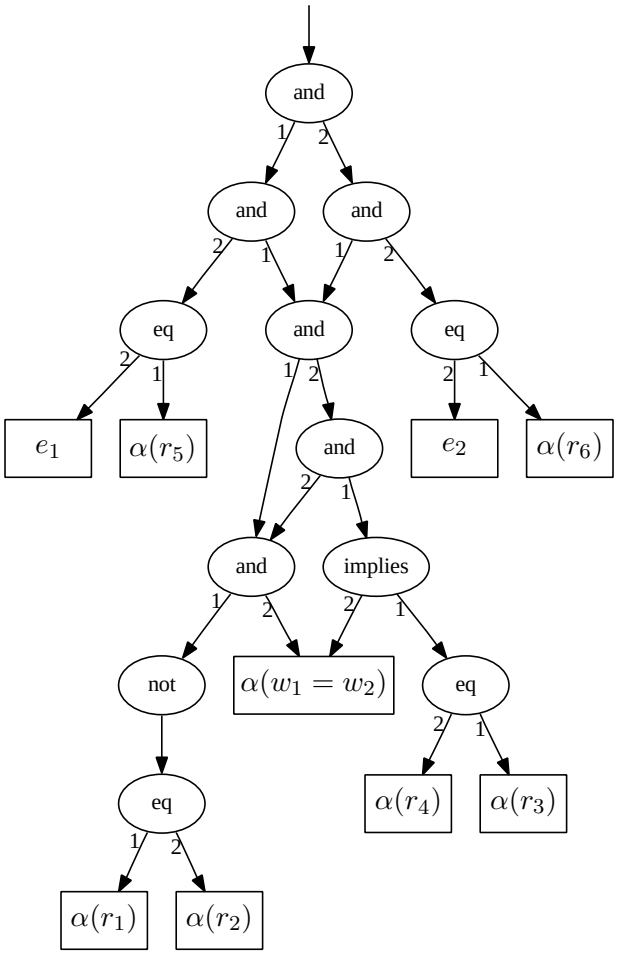


**Figure 12.** Formula $\alpha(\pi)$ for examples 5 and 6.

**Table 1.** Updates for $\chi$ while propagating $read(b, i)$ from source $s$ to destination $d$ under propagation condition $\Delta$. Rule $I$ is a special case used for initialization.

| Rule | s | d | $\Delta$ |
|---|---|---|---|
| $I$ | $b$ | $b$ | $\top$ |
| $D$ | $write(a, j, e)$ | $a$ | $i \neq j$ |
| $U$ | $a$ | $write(a, j, e)$ | $i \neq j$ |
| $R$ | $a$ | $c$ | $a = c$ |
| $L$ | $c$ | $a$ | $a = c$ |

**U:** Let $\psi$ be $\chi(a, read(b,i))$ and $\Delta$ be $i \neq j$. From the induction invariant we know $T_A \models \psi \Rightarrow read(a,i) = read(b,i)$. From (A2) we obtain $\Delta \Rightarrow read(a,i) = read(write(a,j,e),i)$. Therefore, we can conclude $T_A \models \psi \wedge \Delta \Rightarrow read(a,i) = read(write(a,j,e),i) = read(b,i)$.

**R:** Let $\psi$ be $\chi(a, read(b,i))$ and $\Delta$ be $a = c$. From the induction invariant we know $T_A \models \psi \Rightarrow read(a,i) = read(b,i)$. From (A4) we obtain that $\Delta \Rightarrow read(a,i) = read(c,i)$. Therefore, we can conclude $T_A \models \psi \wedge \Delta \Rightarrow read(a,i) = read(b,i) = read(c,i)$.

**L:** can be shown analogously to $R$.

□

**Proposition 8.1.** $T_A \models i = k \wedge \chi(a, read(b,i)) \wedge \chi(a, read(c,k)) \Rightarrow read(b,i) = read(c,k)$

*Proof.* Let $\psi_b$ be $\chi(a, read(b,i))$ and $\psi_c$ be $\chi(a, read(c,k))$. From lemma 8.1 we obtain $T_A \models \psi_b \Rightarrow read(b,i) = read(a,i)$ resp. $T_A \models \psi_c \Rightarrow read(c,k) = read(a,k)$. From (A1) we obtain that $i = k \Rightarrow read(a,i) = read(a,k)$. Therefore, $T_A \models i = k \wedge \psi_1 \wedge \psi_2 \Rightarrow read(b,i) = read(a,i) = read(a,k) = read(c,k)$ follows. □

Thus, each lemma $l$ is $T_A$-valid: $T_A \models l$. Therefore, refining the formula abstraction with these lemmas does not affect satisfiability. Now, can prove that our approach is sound:

**Proposition 8.2.** *If $DP_A(\phi) = unsatisfiable$, then $\phi$ is unsatisfiable modulo $T_A \cup T_B$.*

*Proof.* From proposition 4.1 we obtain that $\phi$ is equisatisfiable to $\pi$, and from proposition 5.1 we know that $\alpha(\pi)$ is an over-approximation of $\pi$. From proposition 8.1 we obtain that each lemma added as formula refinement does not affect satisfiability as it is $T_A$-valid. Thus, if $DP_B$ concludes that $\alpha(\pi)$ is unsatisfiable, then it is sound that our overall decision procedure $DP_A$ concludes that $\phi$ is unsatisfiable. □

## 9. Completeness

In order to show completeness, we define models for terms of sort *Array* and show that they respect semantics of $T_A$. In particular, we show that whenever $DP_B$ finds a $B$-model and the consistency checker can not find an inconsistency, the $B$-model in combination with $\rho$ can be canonically extended to an $A$-model. For the rest of this section we assume that $DP_B$ found a $B$-model and consistency checking terminated without violations of rule $C$.

First of all, we generalize $\sigma$ for terms of sort *Base*. Let $t$ be a term of sort *Base*:

$$\sigma(t) = \sigma(\alpha(t))$$

This means whenever we want the satisfying assignment to a term of sort *Base*, we obtain the assignment to its abstraction. Recall that for terms of sort *Base*, only reads are mapped to abstraction variables.

Now, we define models for terms of sort *Array*, which we also call $\sigma$ in the following. First of all, we need exactly one arbitrary but fixed constant value of sort *Base*. In the

following, we denote this value by 0. Let $a$ be an arbitrary term of sort *Array*, and let $\nu$ be an arbitrary constant value of sort *Base*:

$$\sigma(a)(\nu) = \begin{cases} \sigma(\mathit{read}(b,i)) & \text{if exists } \mathit{read}(b,i) \in \rho(a) \text{ with } \sigma(i) = \nu \\ 0 & \text{otherwise} \end{cases}$$

This means that if we want to read the concrete value of a term $a$ of sort *Array* at index $\nu$, then we have to examine $\rho(a)$. If there is a read $\mathit{read}(b,i)$ where $\sigma(i) = \nu$, then $\sigma(a)(\nu)$ is $\sigma(\mathit{read}(b,i))$. Otherwise, the result is our constant value 0. The constant value 0 is used as default value for array elements which defines our model for all terms, even for those not appearing in $\pi$.

First of all, we show that our definition of array models is well-defined. Let $a$ be a term of sort *Array*, and let $\nu$ be a constant of sort *Base*:

**Proposition 9.1.** *Array model $\sigma$ is well-defined.*

*Proof.* The proof is by cases. First, assume that there exists $\mathit{read}(b,i) \in \rho(a)$ such that $\sigma(i) = \nu$. Let $\mathit{read}(c,j)$ be another read $\in \rho(a)$ such that $\sigma(i) = \sigma(j) = \nu$, then $\sigma(\mathit{read}(b,i)) = \sigma(\mathit{read}(c,j))$. Otherwise, the adapted congruence would be violated. Thus, $\sigma(a)(\nu) = \sigma(\mathit{read}(b,i)) = \sigma(\mathit{read}(c,j))$, which is well-defined.

Now, assume that there does not exist a $\mathit{read}(b,i) \in \rho(a)$ such that $\sigma(i) = \nu$. Then, $\sigma(a)(\nu) = 0$ which is well-defined. □

**Proposition 9.2.** *Array model $\sigma$ respects read semantics of $T_A$.*

*Proof.* Let $a$ be a term of sort *Array*, and let $i$ be a term of sort *Base*. Let $\mathit{read}(a,i)$ occur in $\pi$, then rule $I$ guarantees $\mathit{read}(a,i) \in \rho(a)$. By definition, $\sigma(a)(\sigma(i)) = \sigma(\mathit{read}(a,i))$ and thus $\sigma$ respects read semantics on arrays. □

**Proposition 9.3.** *Array model $\sigma$ respects write semantics of $T_A$.*

*Proof.* In particular, we show that $\sigma$ respects write semantics on arrays. Let $a$ be a term of sort *Array*, and let $i$, $j$ and $e$ be terms of sort *Base*. Let $\mathit{write}(a,i,e)$ occur in $\phi$ resp. $\pi$. Let $\nu$ be a constant value of sort *Base*. Again, the proof is by cases.

First, assume that $\sigma(i) = \nu$. We know that preprocessing step 2 adds the top level constraint $\mathit{read}(\mathit{write}(a,i,e),i) = e$. Therefore, $\mathit{read}(\mathit{write}(a,i,e),i) \in \rho(\mathit{write}(a,i,e))$. Since $\sigma$ is a $B$-model, $\sigma(\mathit{read}(\mathit{write}(a,i,e),i)) = \sigma(e)$. Therefore, $\sigma(\mathit{write}(a,i,e))(\nu) = \sigma(\mathit{read}(\mathit{write}(a,i,e),i)) = \sigma(e)$ which obviously respects update semantics of writes.

Second, assume that $\sigma(i) \neq \nu$. Assume that there exists a $\mathit{read}(b,j)$ with $\sigma(j) = \nu$ in $\rho(\mathit{write}(a,i,e))$. As $\sigma(j) = \nu$ and $\sigma(i) \neq \nu$, rule $D$ guarantees that $\mathit{read}(b,j) \in \rho(a)$. Therefore, $\sigma(\mathit{write}(a,i,e))(\nu) = \sigma(\mathit{read}(b,j)) = \sigma(a)(\nu)$. Analogously, in addition to $\sigma(i) \neq \nu$, assume that there exists a $\mathit{read}(b,j)$ with $\sigma(j) = \nu$ in $\rho(a)$. As $\sigma(j) = \nu$ and $\sigma(i) \neq \nu$, rule $U$ guarantees that $\mathit{read}(b,j) \in \rho(\mathit{write}(a,i,e))$. Therefore, $\sigma(a)(\nu) = \sigma(\mathit{read}(b,j)) = \sigma(\mathit{write}(a,i,e))(\nu)$.

Finally, assume that there does not exist a $\mathit{read}(b,j)$ with $\sigma(j) = \nu$ in $\rho(\mathit{write}(a,i,e))$ resp. $\rho(a)$. Then, $\sigma(\mathit{write}(a,i,e))(\nu) = 0 = \sigma(a)(\nu)$. □

**Proposition 9.4.** *Array model $\sigma$ respects extensional semantics of $T_A$.*

*Proof.* Let $a$ and $c$ be terms of sort *Array* with $a = c$ in $\phi$. We have to show:

$$\sigma(a = c) \quad \Leftrightarrow \quad \forall \nu \, (\sigma(a)(\nu) = \sigma(c)(\nu))$$

The proof is by cases. First, we show:

$$\sigma(a \neq c) \quad \Rightarrow \quad \exists \nu \, (\sigma(a)(\nu) \neq \sigma(c)(\nu))$$

For each equality $a = c$ between terms of sort *Array*, preprocessing step 1 adds the constraint $a \neq c \Rightarrow \mathit{read}(a, \lambda) \neq \mathit{read}(c, \lambda)$. If $DP_B$ finds a model, then there must be an assignment to $\lambda$ such that $\sigma(\mathit{read}(a, \lambda)) \neq \sigma(\mathit{read}(c, \lambda))$, and therefore $\sigma(a)(\nu) \neq \sigma(c)(\nu)$ with $\nu = \sigma(\lambda)$.

Now, we show:

$$\sigma(a = c) \quad \Rightarrow \quad \forall \nu \, (\sigma(a)(\nu) = \sigma(c)(\nu))$$

Assume that there exists a $\mathit{read}(b, j)$ with $\sigma(j) = \nu$ in $\rho(a)$. Rule $R$ guarantees that if $\sigma(a = c) = \top$, then $\mathit{read}(b, j)$ is also in $\rho(c)$. Therefore, $\sigma(a)(\nu) = \sigma(\mathit{read}(b, j)) = \sigma(c)(\nu)$.

Analogously, assume that there exists a $\mathit{read}(b, j)$ with $\sigma(j) = \nu$ in $\rho(c)$. Rule $L$ guarantees that if $\sigma(a = c) = \top$, then $\mathit{read}(b, j)$ is also in $\rho(a)$. Therefore, $\sigma(c)(\nu) = \sigma(\mathit{read}(b, j)) = \sigma(a)(\nu)$.

Finally, assume that there does not exist a $\mathit{read}(b, j)$ in $\rho(a)$ resp. $\rho(c)$. Then, $\sigma(a)(\nu) = 0 = \sigma(c)(\nu)$. □

**Proposition 9.5.** *$DP_A$ is terminating.*

*Proof.* In order to prove overall termination we show that only finitely many lemmas can be generated, and each loop iteration rules out at least one lemma from being regenerated. In the proof of proposition 10.1 we show that the number of lemmas is bounded, i.e. only finitely many lemmas can be generated.

Added lemmas can not be regenerated as for each lemma $\alpha(\mathit{lemma}(\pi, \sigma))$ added in the inner loop, $\sigma(\alpha(\mathit{lemma}(\pi, \sigma))) = \bot$. In future calls to the decision procedure $DP_B$, returning a satisfying assignment, this lemma has to be satisfied, and can thus not be regenerated. □

Now we are able to prove that our approach is complete.

**Proposition 9.6.** *If $\phi$ is unsatisfiable modulo $T_A \cup T_B$, then $DP_A(\phi) = \mathit{unsatisfiable}$.*

*Proof.* We prove the contrapositive. From proposition 9.2, 9.3 and 9.4 we obtain that if $DP_B$ finds a $B$-model $\sigma$ for $\alpha(\pi)$ and the consistency checker does not find a conflict, then this model can always be canonically extended to an $A$-model of $\pi$. From proposition 4.1 we obtain that $\phi$ is equisatisfiable to $\pi$, and obviously, $\sigma$ is also a model of $\phi$. Therefore, in combination with proposition 9.5, which shows that $DP_A$ is terminating, our approach is complete. □

## 10. Complexity

The complexity of our approach depends both on the complexity of the consistency checking algorithm and the complexity of our abstraction refinement, i.e. the upper bound on number of lemmas that have to be generated. We analyze both in the next two subsections.

### 10.1 Consistency Checking

The complexity of the consistency checking algorithm is quadratic in the size of $\pi$, measured in the number of rule applications that update $\rho$. The worst case occurs if each read is propagated to each term of sort *Array*. In the following, we assume consistency checking is performed on-the-fly, i.e. we interleave consistency checking with read propagation. To be precise, whenever we propagate a read to its next destination $d$, we check if there is already a read in $\rho(d)$ that has the same assignment to its index. If this is the case, we immediately perform the consistency check.

Each rule application requires at most one update of $\rho(d)$ and one check according to rule $C$. One check is enough as each read in $\rho(d)$ with the same assignment to its index must have the same assignment to its read value. Otherwise, the consistency checker would have found a conflict before. Therefore, it is sufficient to compare the propagated read with only one representative read. See section 11.2 for a more detailed discussion.

### 10.2 Upper Bound on Number of Lemmas

**Proposition 10.1.** *The number of lemmas is bounded by* $\mathcal{O}(n^2 \cdot 2^n)$ *with* $n = |\phi|$.

*Proof.* Let $r$ be the number of reads in $\phi$, $w$ be the number of writes in $\phi$, and $q$ be the number of equalities between terms of sort *Array* in $\phi$. Then, $\pi$ has $s = r + w + 2q$ reads, and the same number of writes and equalities between terms of sort *Array*.

By checking $C$ on-the-fly in each update to $\rho$, we can make sure that propagation paths do not overlap. Therefore, writes from which the indices stem are all different. Therefore, each lemma consists of at most $w$ comparisons of read and write indices, at most $q$ boolean literals, exactly one read/read index comparison, and exactly one read/read value comparison. There are exactly two read indices in each lemma. Finally, each array equality contributes at most one literal. Therefore, the number of different lemmas is bounded by

$$2^q \cdot 2^w \cdot s^2 = 2^{q+w} \cdot s^2 \leq 2^n \cdot s^2 = \mathcal{O}(n^2 \cdot 2^n)$$

using $q + r + w \leq n = |\phi|$ and $s = \mathcal{O}(n)$. □

Note that rules $L$ resp. $R$ add abstraction variables for equalities between terms of sort *Array* to a lemma only if they have been assigned to $\top$ by $DP_B$. Therefore, they can never occur negatively. Either a boolean abstraction variable occurs positively or not at all.

## 11. Implementation Details

We implemented $DP_A$ in our SMT solver Boolector [20]. Boolector is an efficient SMT solver for the quantifier-free theories of bit-vectors and arrays. Furthermore, it may also be used as model checker for safety properties in the context of bit-vectors and arrays [21]. PicoSAT [11] is used as backend in $DP_B$.

Boolector entered the SMT competition SMT-COMP'08 [6] for the first time. It participated in the quantifier-free theory of bit-vectors `QF_BV`, and in the quantifier-free theory of bit-vectors, arrays and uninterpreted functions `QF_AUFBV`, and won both. In `QF_AUFBV` Boolector solved 16 formulas more than Z3.2, 64 more than last year's winner Z3 0.1 [29], and 103 more than CVC3 [10] version 1.5. In the following, we discuss implementation

details and optimizations in the context of lemmas on demand for the extensional theory of arrays.

### 11.1 Lemma Minimization

Often, lemmas constructed by our approach can be further minimized. In general, there may be many different propagation paths that lead to a theory inconsistency. Starting from the original arrays on which the reads are applied, we respectively perform a breadth-first search to the array at which the inconsistency has been detected. The breadth-first search guarantees that we respectively select one of shortest propagation paths. In general, this leads to stronger lemmas as the propagation conditions are directly used in the premises.

Furthermore, sometimes the premise of a lemma may be even more reduced as some parts may be already subsumed by other parts. For example, let $r_1$ be $\mathit{read}(\mathit{write}(a, j, e_1), i_1)$, and $r_2$ be $\mathit{read}(\mathit{write}(a, j, e_2), i_2)$. Consider the following formula:

$$i_1 \neq j \ \wedge \ i_2 \neq j \ \wedge \ r_1 \neq r_2$$

Let us assume that $DP_B$ generates an assignment such that $\sigma(i_1) \neq \sigma(j)$, $\sigma(i_2) \neq \sigma(j)$, $\sigma(i_1) = \sigma(i_2)$ and $\sigma(r_1) \neq \sigma(r_2)$. The consistency checker propagates $r_1$ and $r_2$ down to $a$ as $\sigma(i_1) \neq \sigma(j)$ and $\sigma(i_2) \neq \sigma(j)$. It detects a conflict as $\sigma(i_1) = \sigma(i_2)$, but $\sigma(r_1) \neq \sigma(r_2)$ and generates the following lemma:

$$i_1 \neq j \ \wedge \ i_2 \neq j \ \wedge \ i_1 = i_2 \quad \Rightarrow \quad r_1 = r_2$$

This lemma can be minimized as one inequality is already subsumed by the other in combination with the equality of the read indices. For example, it can be minimized to:

$$i_1 \neq j \ \wedge \ i_1 = i_2 \quad \Rightarrow \quad r_1 = r_2$$

If we encode inequality and equality into CNF, then the shorter lemma results in fewer clauses than in the original case. This may be beneficial for the runtime of the SAT solver.

In general, the indices that are used to decide whether the read should be propagated or not, e.g. $j$ in the previous example, can be uniquely hashed. Each lemma contains $\mathit{index}(\mathit{read}_1) = \mathit{index}(\mathit{read}_2)$ in the premise. Therefore, it is sufficient to encode that either $\mathit{index}(\mathit{read}_1)$ or $\mathit{index}(\mathit{read}_2)$ is unequal to write indices that occur in both propagation paths.

### 11.2 Implementing $\rho$

The set of reads $\rho$ can be efficiently implemented by hashtables. For each term of sort *Array* a hashtable is created. A concrete assignment to a read index can be used as hash value. In the following, we assume consistency checking is performed on-the-fly, i.e. we interleave consistency checking with read propagation. Whenever we propagate a read to its next destination $d$, we check if there is already a read in $\rho(d)$, which has the same assignment to its index. If this is the case, we immediately perform the consistency check.

In general, for each term $a$ of sort *Array* it is sufficient to maintain exactly one read per concrete assignment to its read index in $\rho(a)$. If we have multiple reads in $\rho(a)$ that have the same assignment to the index values and the same assignment to the read values,

then we can think of them as one equivalence class. In particular, they must have the same assignment to the read values. Otherwise, on-the-fly consistency checking would have found a conflict before. Therefore, it is sufficient to remember and propagate only one representative per class. Whenever we propagate one particular read $r_1$ to its next destination $d$ and there is already a read $r_2$ in $\rho(d)$ which has the same assignment to index value and read value, then we do not have to insert $r_1$ into $\rho(d)$ as we already have the representative $r_2$ in $\rho(d)$. Therefore, we can skip the current propagation path of $r_1$ as we already have the representative $r_2$ in $\rho(d)$ which is also propagated further if necessary.

Implementing $\rho$ by hashtables results in fewer propagations and consistency checks. In particular, in each propagation, consistency checking must only be performed with one representative instead of all propagated reads that have the same assignment to the index values and the same assignment to the read values.

### 11.3 Positive Array Equalities

Whenever the boolean structure of the original formula prevents $DP_B$ from setting the boolean abstraction variable of an array equality to $\bot$, then preprocessing step 1 can be skipped. Recall that preprocessing step 1 adds one pair of virtual reads as witness for array inequality. However, if $DP_B$ can not set the abstraction variable to $\bot$, then the virtual reads do not have to be introduced. Rules $L$ and $R$ are sufficient to enforce extensional consistency. The virtual reads can be safely omitted, which is beneficial for $DP_A$ as fewer reads have to be propagated and checked. Furthermore, this decreases the number of potential lemmas. As an example for a situation where preprocessing step 1 can be skipped, think of a formula where the boolean structure is in CNF and each array equality occurs only positively. An example where preprocessing step 1 can be omitted is shown in Fig. 13.

### 11.4 Treating Writes as Reads

Inside Boolector we use a polymorphic node type called *access*. A node of this type is either a read or a write. The main idea of this implementation technique is that we can treat a write as read on itself. If we have a write $write(a, i, e)$, then (A2) guarantees that a read at index $i$ must have read value $e$. Instead of explicitly introducing these reads by preprocessing step 2, we can interpret a write as read on itself. We use the polymorphic functions *index* and *value* on nodes of type *access*. If we use them on a node representing a read, they return read index resp. read value. However, if we use them on a node representing $write(a, i, e)$, they return write index $i$ resp. write value $e$.

Let $w$ be $write(a, i, e)$ and let $r$ be $read(w, i)$. If we apply *index* to $r$ , we obtain $i$ which is the same as applying the function directly on $w$. If we apply *value* to $r$, we get a read value which has to be equal to $e$. However, instead of introducing an abstraction variable for $r$ and refining it to $e$, we can use $e$ directly, which is what we obtain if we apply *value* to $w$ directly.

In our implementation, we do not actually propagate reads. However, we propagate objects of type *access* which are either reads or writes. In this way, we also ensure congruence on write values. Therefore, we can completely skip preprocessing step 2.

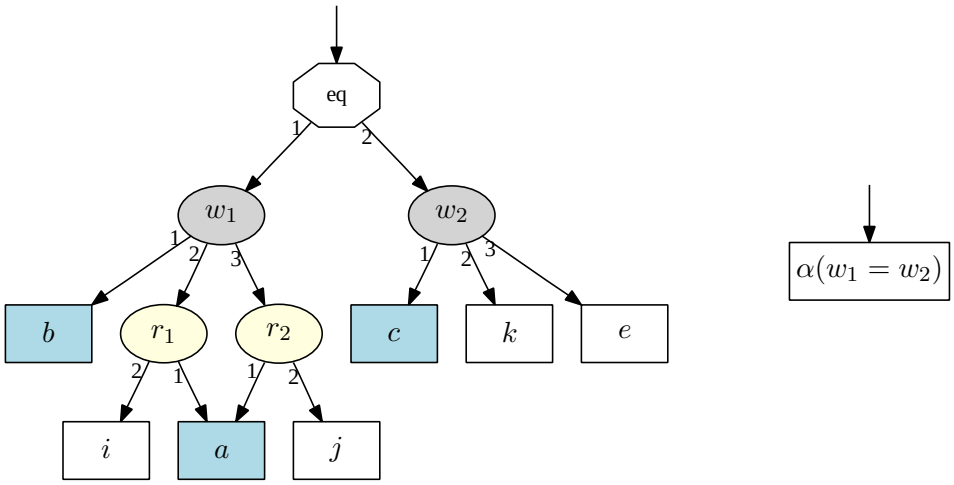


**Figure 13.** Formula $\phi$ resp $\pi$ for example 7 is shown left. Preprocessing step 1 is omitted as the equality between $w_1$ and $w_2$ occurs only positively. Preprocessing step 2 is omitted as it is not necessary if writes are treated and propagated as reads. The abstraction $\alpha(\pi)$ is shown right. The abstraction variables $\alpha(r_1)$ and $\alpha(r_2)$ are not shown as they are not reachable from the root.

**Example 7.**

*Let $\phi$ resp. $\pi$ be as shown left in Fig. 13. The abstraction $\alpha(\pi)$ is shown right.*

*We run $DP_B$ on $\alpha(\pi)$ and assume that $DP_B$ generates a model $\sigma$ such that $\sigma(\alpha(w_1 = w_2)) = \top$, $\sigma(\alpha(r_1)) = \sigma(k)$, and $\sigma(\alpha(r_2)) \neq \sigma(e)$.*

*We perform on-the-fly consistency checking in depth-first search manner. Initially, $\rho(a) = \{r_1, r_2\}$, $\rho(w_1) = \{w_1\}$, and $\rho(w_2) = \{w_2\}$ We treat $w_1$ as read and propagate it to $w_2$ as $\sigma(\alpha(w_1 = w_2)) = \top$. We update $\rho(w_2)$ to $\{w_1, w_2\}$. We check $\rho(w_2)$, find an inconsistency according to rule C, as $\sigma(\alpha(r_1)) = \sigma(k)$, but $\sigma(\alpha(r_2)) \neq \sigma(e)$. We generate the following lemma:*

$$r_1 = k \ \wedge \ w_1 = w_2 \quad \Rightarrow \quad r_2 = e$$

### 11.5 Synthesis on Demand

Boolector uses a functional representation for bit-vectors. Each term of type bit-vector is mapped to a vector of And-Inverter Graphs (AIGs) [41]. For example, a term that represents the multiplication of two 32 bit-vectors is mapped to a multiplication circuit represented by AIGs. Each AIG of the resulting AIG vector represents one output of the circuit. During the construction of the AIGs, local two-level rewriting is performed [18]. The AIG vectors are encoded into CNF and incrementally added to the SAT solver.

In principle, each term could be encoded into AIG vectors and then encoded into CNF up front. However, this is not always necessary as a theory conflict which leads to unsat-

isfiability may occur in one small part of the formula. For example, consider the following formula. Let us assume that $i, j, x$ and $y$ are bit-vectors with bit-width 64 and *udiv* represents unsigned bit-vector division:

$$i \neq j \ \wedge \ read(write(a, j, udiv(x, y)), i) \neq read(a, i)$$

As $i \neq j$, the two reads have to be equal according to (A1). Hence, this formula is unsatisfiable regardless of the unsigned division. If we synthesize AIGs resp. encode to CNF up front, then the SAT solver will unnecessarily have to handle a complex 64 bit division circuit on the CNF layer, although unsatisfiability can be easily concluded without it. Therefore, we postpone AIG synthesis resp. CNF encoding of read indices, write indices, and write values. As recently as the consistency checker has to examine the concrete bit-vector assignment, the synthesis and encoding is performed, and the SAT solver is called incrementally.

In [25] for each call to the SAT solver the CNF is generated from scratch. We incrementally add clauses as required. The incremental usage of the SAT solver follows [26, 32] as implemented in MiniSAT [33] and PicoSAT [11]. Lazy synthesis is applied in our array consistency checking algorithm if we find a read or write operation, for which the bit-vector arguments have not been synthesized yet. At this point the SAT solver already determined a model for previously added clauses. This boolean model needs to be extended to new CNF variables and clauses synthesized lazily for those unsynthesized read or write indices and write values. These indices and values are unconstrained at this point and thus the SAT solver should always be able to extend the model.

However, due to restarts [39] even if phase saving [11, 47] is employed, the SAT solver is free to flip a value of the original model. If this happens, the array consistency algorithm has to be aborted and restarted from scratch: the read and write propagations performed up to this point may have become invalid. In our experiments this rarely happens, simply because phase saving tries to keep the old model. However, ignoring changed values in the old model would render the array consistency checking algorithm incomplete and unsound.

In our original version we simply restarted the array consistency checking algorithm as soon as new indices or write values have been synthesized lazily. The overhead can be avoided by adding a new API function to the SAT solver that determines if an incremental call to the SAT solver has changed the model generated in a previous SAT-call to the SAT solver. This can be implemented in a conservative way by monitoring forced assignments of variables. If a forced assignment assigns a value to a variable which is different from the previously saved assigned value to this variable, and this variable is not a new synthesis variable, then the old model has changed. In the current implementation we conservatively mark the old model as changed, even if later in the same call to the SAT solver these changes are reverted.

It would also be possible to precisely determine whether the old model has changed by saving the old model value instead of misusing saved phases. Another option is to ask the SAT solver to search for a model that extends the previous model. Both alternatives are more complex to implement, and we leave it for future work to determine whether they are more effective than our current solution.

### 11.6 CNF Encoding

In Boolector lemmas are directly added on the CNF level. No additional terms have to be created. Therefore, we need an efficient way to encode lemmas with bit-vector equalities and inequalities to SAT. For example, assume the congruence axiom (A1) is violated and we add the bit-vector lemma $i = j \Rightarrow v = w$. In order to encode this lemma to CNF, we introduce a fresh boolean variable $e$:

$$(i = j \Rightarrow e) \wedge (e \Rightarrow v = w)$$

This formula is equisatisfiable and can be rewritten into $(i \neq j \vee e) \wedge (\bar{e} \vee v = w)$. Let $m$ be the number of bits of $i$ and $j$, and let $n$ be the number of bits of $v$ and $w$. We introduce $m$ fresh variables $d_k$ and encode $i \neq j$ as follows:

$$\bigwedge_{k=1}^{m} \left( (i_k \vee j_k \vee \bar{d}_k) \wedge (\bar{i}_k \vee \bar{j}_k \vee \bar{d}_k) \right)$$

If $i_k = j_k$, then $d_k$ is forced to $\bot$. However, if $i_k \neq j_k$, then $d_k$ is unconstrained and can be set to $\top$. In order to encode $v = w$, we add the following clauses:

$$\bigwedge_{k=1}^{n} \left( (\bar{v}_k \vee w_k \vee \bar{e}) \wedge (v_k \vee \bar{w}_k \vee \bar{e}) \right)$$

Finally, we relate the two parts through a *linking clause*:

$$e \vee \bigvee_{k=1}^{m} d_k$$

The idea of this encoding is as follows. If $i \neq j$, then they differ in at least one bit. Therefore, one $d_k$ can be set to $\top$ to satisfy the linking clause. The variable $e$ is now unconstrained and can be set to $\bot$. Therefore, $v$ and $w$ are not forced to be equal. However, if $i = j$, each $d_k$ is $\bot$. In order to satisfy the linking clause, $e$ has to be set to $\top$, which forces $v$ and $w$ to be equal.

In general, each lemma has the following form [2], after eliminating implication:

$$\bigvee_{k=1}^{x} b_k \vee \bigvee_{k=1}^{y} (u_k \neq v_k) \vee \bigvee_{k=1}^{z} (s_k = t_k)$$

Assuming all bit-vectors have $n$ bits, the CNF has $2 \cdot n \cdot (y + z) + 1$ clauses and $y \cdot n + z$ fresh boolean variables. The clauses are all ternary, except the linking clause of size $x + y \cdot n + z$.

In principle, bit-vector equalities and inequalities can be natively supported by SAT solvers, similar as in [12]. All different constraints are directly supported by PicoSAT. This approach can simplify implementation complexity of SMT solvers that support bit-vectors and is scheduled as future work.

2. Our lemmas have exactly one inequality ($y = 1$), but we discuss the general case.

## 12. Experiments

We compared Boolector version 0.8 to STP version 0.1-11-18-2008. Recall that this article is an improvement and extension of a preliminary version presented at the SMT'08 workshop [19]. We do not repeat our experiments in [19] and we also do not repeat the results of the SMT competition SMT-COMP'08 [6], where Boolector clearly won the division of the quantifier-free theory of bit-vectors, arrays and uninterpreted functions `QF_AUFBV`, and the quantifier-free theory of bit-vectors `QF_BV`. Nevertheless, as Boolector and STP have similar approaches we provide additional experiments comparing the performance of Boolector to the performance of STP. See the related work section 13 for a comparison between the approaches of Boolector and STP.

STP does not support equalities between arrays. However, many interesting and challenging array benchmarks in the SMT-LIB [9] compare arrays for equality, i.e. are extensional. Furthermore, STP reads the CVC Lite [5] format [1] while Boolector reads BTOR [21] and SMT-LIB format [48]. This was a problem as formula conversion with the help of CVC3 was not always possible. Therefore, we had to restrict our experiments to non-extensional examples that were either available both in SMT-LIB and in CVC format or just in one format and conversion to the other format was possible.

We selected the STP benchmarks that are available in `QF_AUFBV` in the SMT-LIB [9]. Unfortunately, `testcase10` and `testcase15` are not available. We omitted two `cksumcookie` benchmarks as they were trivial for Boolector and STP. Furthermore, we selected seven benchmarks representing formal verification of the bubble sort algorithm for array elements of 32 bit. The number within the name of the benchmark represents the size of the array. These benchmarks are also part of the SMT-LIB and can be found in `QF_AUFBV`.

We ran our benchmarks on our cluster of 3 GHz Pentium IV with 2 GB main memory, running Ubuntu Linux. We set a time limit of 1800 seconds and a memory limit of 1500 MB. The results are shown in table 2. The memory is shown in MB and the time in seconds. Boolector clearly outperforms STP [3].

## 13. Related Work

Ganesh et al. present a decision procedure for the theory of bit-vectors and arrays [36], but without support for equalities on arrays. Similar to our approach, they use word-level preprocessing and an abstraction refinement loop. They implemented their decision procedure in STP, which is an SMT solver optimized for large problems in software analysis applications. Unfortunately, their decision procedure is presented without details. Neither in [35] nor in [36] are details on their abstraction refinement and lazy axiom instantiations presented. Neither do they prove soundness nor completeness.

Ganesh points out in [35] that efficiently handling nested writes is crucial for SMT solvers applied to software verification. An eager read-over-write elimination applied to deeply nested writes creates a new copy of the DAG of writes for every distinct read index.

3. Note that STP can not decide satisfiability for benchmark `noregions-fullmemite`. After a few seconds it terminates with exit code 0, but does not print out the result. First, we thought that this might be a problem due to the YACC-parser. However, also CVC3 uses a YACC-parser and it can parse the benchmark without problems. Setting the stack size of the STP YACC-parser to unlimited did also not help. Therefore, it remains unclear why STP terminates without a result.

**Table 2.** Experimental comparison between STP and Boolector.

| | | STP | | Boolector | |
|---|---|---|---|---|---|
| Benchmark | Status | Mem | Time | Mem | Time |
| 610dd9dc | sat | out of memory | out of memory | **9** | **34** |
| blaster-concrete | unsat | 48 | 2 | **23** | **0** |
| blaster-small | sat | 2 | 0 | **0** | 0 |
| blaster | unsat | 85 | 5 | **31** | **0** |
| blaster-wp-13 | unsat | 48 | 2 | **30** | **0** |
| blaster-wp-4 | unsat | 151 | 10 | **38** | **1** |
| blaster-wp-8 | unsat | 68 | 4 | **30** | **0** |
| bubsort005un | unsat | 2 | 4 | **1** | **0** |
| bubsort006un | unsat | 3 | 30 | **1** | **1** |
| bubsort007un | unsat | 6 | 151 | **1** | **8** |
| bubsort008un | unsat | 11 | 784 | **2** | **26** |
| bubsort009un | unsat | out of time | out of time | **2** | **72** |
| bubsort010un | unsat | out of time | out of time | **4** | **325** |
| bubsort012un | unsat | out of time | out of time | out of time | out of time |
| cmu-model15 | sat | 11 | 2 | **5** | **0** |
| cmu-model16 | sat | 11 | 2 | **5** | **0** |
| cmu-model17 | sat | 11 | 2 | **4** | **0** |
| ff | unknown | out of memory | out of memory | out of memory | out of memory |
| grep0065 | unsat | 43 | 4 | **2** | **0** |
| grep0084 | sat | 183 | 116 | **18** | **15** |
| grep0095 | sat | 198 | 127 | **19** | **17** |
| grep0106 | sat | 175 | 116 | **20** | **18** |
| grep0117 | sat | 193 | 115 | **20** | **19** |
| grep0777 | sat | out of memory | out of memory | **33** | **99** |
| noregions-fullmemite | unknown | no result | no result | **380** | **40** |
| noregions-stpmem | unknown | out of time | out of time | **106** | **1526** |
| testcase01 | sat | **22** | **2** | 25 | 52 |
| testcase02 | sat | 466 | 50 | **413** | **14** |
| testcase03 | sat | 574 | 66 | **500** | **18** |
| testcase04 | sat | 590 | 69 | **514** | **18** |
| testcase05 | sat | 589 | 68 | **513** | **18** |
| testcase06 | unsat | **297** | 31 | 490 | **13** |
| testcase07 | unsat | **298** | 32 | 491 | **13** |
| testcase08 | unsat | **293** | 31 | 490 | **12** |
| testcase09 | sat | 584 | 68 | **513** | **18** |
| testcase11 | sat | **323** | 15 | 515 | **13** |
| testcase12 | sat | **346** | 16 | 550 | **15** |
| testcase13 | sat | **67** | 5 | 79 | **2** |
| testcase14 | sat | 63 | 28 | **58** | **1** |
| testcase16 | sat | 952 | **50** | **318** | 75 |
| testcase17 | sat | 347 | 23 | **335** | **12** |
| testcase18 | sat | 43 | **4** | **30** | 87 |
| testcase19 | sat | 28 | 3 | 28 | 3 |
| testcase20 | sat | **393** | **45** | 505 | 117 |
| testcase21 | sat | **390** | **35** | 520 | 53 |
| thumbnail-spin1 | sat | 1116 | 75 | **966** | **42** |

This eager rewriting blows up in space and is therefore not appropriate, which is also confirmed by our experiments in [19]. Similar to our approach, Ganesh et al. handle read over writes lazily. Unfortunately, it remains unclear how their refinement actually works. In [35] they give a hint that they use the following policy in their refinement. If they find a spurious model, then they take a non-constant array index term for which at least one array axiom is violated, and add all of the violated axioms involving that term. This is in contrast to our approach, where we always add exactly one lemma.

It is difficult to compare STP to Boolector as STP does not support equalities between arrays. Many interesting and challenging array benchmarks in the SMT-LIB [9] compare arrays for equality. Therefore, we can not compare the performances of Boolector and STP on those examples. Nevertheless, our experiments on non-extensional examples, i.e. examples without comparing arrays for equality, in section 12 clearly show that Boolector outperforms STP, even on many examples for which STP has been optimized.

Stump et al. describe a decision procedure for an extensional theory of arrays in [51] and discuss earlier work. They present their decision procedure as a proof system and prove its correctness. Their approach works in two phases. In the first phase a set of subgoals is created such that no subgoal contains write expressions, which may blow up in space. This is in contrast to our approach where we do *not* eagerly eliminate write expressions. Finally, the original goal is satisfiable if and only if one of the subgoals is satisfiable. The key concept of their approach is to use partial equations of the following form:

$$a =_I b \quad \Leftrightarrow \quad \forall i.\ i \notin I \rightarrow \mathit{read}(a, i) = \mathit{read}(b, i)$$

The set $I$ is a set of indices on which the arrays do not have to be equal. With these partial equations, write expressions are eliminated in the following way:

$$\mathit{write}(a, i, e) = b \quad \Leftrightarrow \quad a =_{\{i\}} b \ \wedge\ \mathit{read}(b, i) = e$$

Note that the idea of adding the constraint $\mathit{read}(b, i) = e$ to ensure write value consistency is similar to our preprocessing step 2. The complexity of their approach is $\mathcal{O}(2^{N\ lg\ N})$, where $N$ is the size of the original goal.

Bradley et al. introduce the array property fragment in [16]. The main idea of this fragment is that array indices can be universally quantified with some restrictions. The restrictions are necessary to avoid undecidability. Array property fragments allow the expression of bounded and unbounded properties of arrays, e.g. array equality, universal properties, partitioning and sorting. Unlike our decision procedure, their decision procedure has to eliminate write expressions, which may blow up in space. Furthermore, the formula has to be put into negation normal form, which is also not necessary for our approach. Universally quantified subterms are reduced to finite conjunctions. This is done by instantiating quantified variables over a set of index terms. In [17] Bradley et al. prove that simple extensions of the array property fragment, e.g. nested reads, may already result in a fragment for which satisfiability is undecidable. We conjecture that our approach can be extended to handle interesting properties of the array property fragment as well.

In [37] Ghilardi et al. consider extensions of the theory of arrays with extensionality where indices have the algebraic structure of Presburger arithmetic. Their extensions consider characteristics like dimension and sortedness. They integrate available decision pro-

cedures for the theory of arrays and Presburger arithmetic in combination with automatic instantiation strategies.

In [38] Goel et al. introduce a rule-based axiom-instantiating decision procedure for the parametric theory of arrays, where formulas can refer to multiple and arbitrarily nested array types. Like our approach, they use lazy axiom instantiations. Interestingly, theories of arrays are interpreted in the context of updatable functions, i.e. *read* is interpreted as a function application and *write* as an update operator. They extended the congruence-closure module of the SMT solver DPT [2] in order to implement their approach. This is contrast to our approach, which is not based on congruence-closure. Unfortunately, the effectiveness of our implementation can not be compared to their implementation as they only report on experimental results in `QF_AUFLIA`, but not in `QF_AUFBV`.

Manolios et al. introduce a memory abstraction algorithm in the context of the Bit Level Analysis Tool BAT in [43]. Memories are functionally represented with the help of array semantics, i.e. get and set in BAT correspond to read and write in SMT. Their approach uses equivalence classes of memories. Index values, i.e. bit-vector addresses that are used in set and get, are analyzed, and shorter bit-vector addresses for addressing the abstract memory are created. In addition to memory abstraction, rewriting is performed to simplify memory accesses.

In [14] Bofill et al. introduce a write base approach in the context of DPLL(T) to handle the extensional theory of arrays. Their approach does not eliminate write expressions and does not lazily instantiate array axioms. However, the theory solver asks the boolean engine to split on demand on equality literals between indices.

The theorem prover Simplify [31] uses Nelson-Oppen [45] to combine decision procedures for several theories, e.g. arrays, maps and sets, and uses a matcher to reason about quantifiers.

Finally, UCLID [42] is a decision procedure for the theories of uninterpreted functions, integer linear arithmetic, and arrays which supports some limited forms of quantification. UCLID uses eager translation to SAT and lambda expressions for arrays. Furthermore, Bryant et al. propose a decision procedure for bit-vector arithmetic with automatic abstraction refinement in [25]. The procedure is implemented in UCLID. It alternates between under- and over-approximations of the original formula.

## 14. Conclusion

We discussed lemmas on demand for the extensional theory of arrays. In particular, we presented our preprocessing, formula abstraction, consistency checking, and abstraction refinement, i.e. lemma generation on demand. Furthermore, we proved that our approach is sound and complete, and provided a formal description of our abstraction refinement. We discussed complexity and provided implementation details and optimizations that are important for real implementations. Then, we discussed related work. Our overall experiments and the SMT competition in 2008 showed that our approach implemented in our SMT solver Boolector is faster than other state-of-the-art solvers. Similar to model checking where abstraction is used to handle the state explosion problem, abstraction techniques are the key to efficiently decide large and complex SMT formulas.

### 14.1 Acknowledgements

We want to thank Nikolaj Bjørner, Leonardo De Moura, Robert Nieuwenhuis and anonymous reviewers for their helpful comments on preliminary versions.

## References

[1] CVC Lite Documentation, 2008. http://verify.stanford.edu/CVCL/doc/.

[2] Decision Procedure Toolkit, 2008. http://sourceforge.net/projects/dpt.

[3] W. Ackermann. *Solvable Cases of the Decision Problem*. Studies in Logic and the Foundations of Mathematics. North-Holland, 1954.

[4] C. Barrett. *Checking Validity of Quantifier-Free Formulas in Combinations of First-Order Theories*. PhD thesis, Stanford University, 2003.

[5] C. Barrett and S. Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. In *Proc. CAV'04*, pages 515–518. Springer, 2004.

[6] C. Barrett, M. Deters, A. Oliveras, and A. Stump. SMT-Comp, 2008. www.smtcomp.org.

[7] C. Barrett, D. Dill, and A. Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In *Proc. CAV'02*, pages 236–249. Springer, 2002.

[8] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on Demand in SAT Modulo Theories. In *Proc. LPAR'06*, pages 512–526. Springer, 2006.

[9] C. Barrett, S. Ranise, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2008.

[10] C. Barrett and C. Tinelli. CVC3. In *Proc. CAV'07*, pages 298–302. Springer, 2007.

[11] A. Biere. PicoSAT Essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, **4**:75–97, 2008.

[12] A. Biere and R. Brummayer. Consistency Checking of All Different Constraints over Bit-Vectors within a SAT-Solver. In *Proc. FMCAD'08*, pages 223–226. IEEE, 2008.

[13] A. Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*. IOS Press, 2009.

[14] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodriguez-Carbonell, and A. Rubio. A Write-Based Solver for SAT Modulo the Theory of Arrays. In *Proc. FMCAD'08*. IEEE, 2008.

[15] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, S. Ranise, P. van Rossum, and R. Sebastiani. Efficient Satisfiability Modulo Theories via Delayed Theory Combination. In *CAV'05*, pages 335–349. Springer, 2005.

[16] A. Bradley and Z. Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer, 2007.

[17] A. Bradley, Z. Manna, and H. Sipma. What's Decidable About Arrays? In *Proc. VMCAI'06*, pages 427–442. Springer, 2006.

[18] R. Brummayer and A. Biere. Local Two-Level And-Inverter Graph Minimization without Blowup. In *Proc. MEMICS'06*, pages 32–38. Faculty of Information Technology, Brno University, 2006.

[19] R. Brummayer and A. Biere. Lemmas on Demand for the Extensional Theory of Arrays. In *Proc. SMT'08*. ACM, 2008.

[20] R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Proc. TACAS'09*, pages 174–177. Springer, 2009.

[21] R. Brummayer, A. Biere, and F. Lonsing. BTOR: Bit-Precise Modelling of Word-Level Problems for Model Checking. In *Proc. BPR'08*. ACM, 2008.

[22] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani. A Lazy and Layered SMT($\mathcal{BV}$) Solver for Hard Industrial Verification Problems. In *Proc. CAV'07*, pages 547–560. Springer, 2007.

[23] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, A. Santuari, and R. Sebastiani. To Ackermann-ize or Not to Ackermann-ize? On Efficiently Handling Uninterpreted Function Symbols in SMT ($\mathcal{EUF} \cup \mathcal{T}$). In *Proc. LPAR'06*, pages 557–571. Springer, 2006.

[24] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. Delayed Theory Combination vs. Nelson-Oppen for Satisfiability Modulo Theories: A Comparative Analysis. In *Proc. LPAR'06*, pages 527–541. Springer, 2006.

[25] R. Bryant, D. Kroening, J. Ouaknine, S. Seshia, O. Strichman, and B. Brady. Deciding Bit-Vector Arithmetic with Abstraction. *International Journal on Software Tools for Technology Transfer (STTT)*, 2008. To appear.

[26] K. Claessen and N. Sörensson. New Techniques that Improve MACE-style Finite Model Finding. In *CADE-19, Workshop W4, Model Computation – Principles, Algorithms, Applications*, 2003.

[27] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. *Journal of the ACM (JACM)*, pages 752–794, 2003.

[28] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, **5**, 1962.

[29] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proc. TACAS'08*, pages 337–340. Springer, 2008.

[30] L. de Moura and H. Rueß. Lemmas on Demand for Satisfiability Solvers. In *Proc. SAT'02*, pages 244–251. Springer, 2002.

[31] D. Detlefs, G. Nelson, and J. Saxe. Simplify: A Theorem Prover for Program Checking. *Journal of the ACM (JACM)*, **52**:365–473, 2005.

[32] N. Eén and N. Sörensson. Temporal Induction by Incremental SAT Solving. *Electronic Notes in Theoretical Computer Science (ENTCS)*, **89**(4), 2003.

[33] N. Eén and N. Sörensson. An Extensible SAT-Solver. In *Proc. SAT'04*, pages 333–336. Springer, 2004.

[34] C. Flanagan, R. Joshi, and J. Saxe. Theorem Proving Using Lazy Proof Explication. In *Proc. CAV'03*, pages 355–367. Springer, 2003.

[35] V. Ganesh. *Decision Procedures for Bit-Vectors, Arrays and Integers*. PhD thesis, Computer Science Department, Stanford University, 2007.

[36] V. Ganesh and D. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *Proc. CAV'07*, pages 519–531. Springer, 2007.

[37] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Decision Procedures for Extensions of the Theory of Arrays. *Annals of Mathematics and Artificial Intelligence*, **50**:231–254, 2007.

[38] A. Goel, S. Krstic, and A. Fuchs. Deciding Array Formulas with Frugal Axiom Instantiation. In *Proc. SMT'08*. ACM, 2008.

[39] C. Gomes, B. Selman, and H. Kautz. Boosting Combinatorial Search Through Randomization. In *Proc. AAAI'98*, pages 431–437. AAAI Press, 1998.

[40] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 2008.

[41] A. Kühlmann, M. Ganai, and V. Paruthi. Circuit-based Boolean Reasoning. In *Proc. DAC'01*, pages 232–237. ACM, 2001.

[42] S. Lahiri and S. Seshia. The UCLID Decision Procedure. In *Proc. CAV'04*, pages 475–478. Springer, 2004.

[43] P. Manolios, S. Srinivasan, and D. Vroon. Automatic Memory Reductions for RTL Model Verification. In *Proc. ICCAD'06*. ACM, 2006.

[44] J. McCarthy. Towards a Mathematical Science of Computation. In *Proc. IFIP Congress*, pages 21–28. North-Holland, 1962.

[45] G. Nelson and D. Oppen. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **1**:245–257, 1979.

[46] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM (JACM)*, **53**:937–977, 2006.

[47] K. Pipatsrisawat and A. Darwiche. RSat 2.0: SAT Solver Description. Technical Report D–153, Automated Reasoning Group, Computer Science Department, UCLA, 2007.

[48] S. Ranise and C. Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006. Available at www.SMT-LIB.org.

[49] R. Sebastiani. Lazy Satisfiability Modulo Theories. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, **3**:141–224, 2007.

[50] R. Shostak. Deciding Combinations of Theories. *Journal of the ACM (JACM)*, **31**:1–12, 1984.

[51] A. Stump, C. Barrett, D. Dill, and J. Levitt. A Decision Procedure for an Extensional Theory of Arrays. In *Proc. LICS'01*, pages 29–37. IEEE, 2001.

[52] G. Tseitin. On the Complexity of Proofs in Propositional Logics. *Automation of Reasoning: Classical Papers in Computational Logic 1967-1970*, **2**, 1983. Originally published 1970.